PAPER Special Section on Formal Approach

A Formal Verification of a Subset of Information-Based Access Control Based on Extended Weighted Pushdown System

Pablo LAMILLA ALVAREZ[†a)], *Nonmember and* **Yoshiaki TAKATA**[†b)], *Member*

SUMMARY Information-Based Access Control (IBAC) has been proposed as an improvement to History-Based Access Control (HBAC) model. In modern component-based systems, these access control models verify that all the code responsible for a security-sensitive operation is sufficiently authorized to execute that operation. The HBAC model, although safe, may incorrectly prevent the execution of operations that should be executed. The IBAC has been shown to be more precise than HBAC maintaining its safety level while allowing sufficiently authorized operations to be executed. However the verification problem of IBAC program has not been discussed. This paper presents a formal model for IBAC programs based on extended weighted pushdown systems (EWPDS). The mapping process between the IBAC original semantics and the EWPDS structure is described. Moreover, the verification problem for IBAC programs is discussed and several typical IBAC program examples using our model are implemented.

key words: *weighted pushdown systems, access control, model checking*

1. Introduction

Access control models are adopted in modern component-based systems, specifically the Stack-Based Access Control (SBAC) supported in environments such as Java virtual machines [7] and Microsoft .NET Common Language Runtime (CLR). This model uses stack-inspection to verify that all the code responsible for a security-sensitive operation has been granted a given set of permissions P which guards that operation. This is checked using a primitive function called *checkPermission* which checks that the set P is granted on all the callers currently on the execution stack when the security-sensitive operation is tried to be executed. However SBAC may allow *non-trusted code* to influence a security-sensitive operation performed by *trusted code* because SBAC does not retain the security information of previously executed methods because they are not on the execution stack anymore. To solve this problem, Abadi et al. [1] introduced a novel approach called History-Based Access Control (HBAC) which registers the history of all previously executed methods. HBAC systems ensure that all the code previously executed is sufficiently authorized to access a protected resource. However, since not all the code previously executed may have influence on the protected resource, the HBAC is excessively restrictive in some cases. Pistoia et al. formally presented in [2] a novel security model called Information-Based Access Control (IBAC) which has proven to be less restrictive than HBAC while maintaining the level of security assurance of HBAC. The IBAC model tracks not only the permissions of blocks of code but also the dynamic permissions of each variable. Therefore, the IBAC verifies that only the code responsible for a security-sensitive operation is sufficiently authorized.

Since the main purpose of the model proposed in [2] is not formal verification, the model cannot be directly applicable to model checking problems. This paper proposes a formal model for IBAC programs suitable for formal verification. We model the behavior of an IBAC program as an extended weighted-pushdown system (EWPDS) [5]. This extension of pushdown systems (PDS) naturally models the behavior of an IBAC program in which subsets of permissions are maintained and altered at every procedure call and return. Moreover, in some cases using EWPDS improves the efficiency of model-checking [4] compared with a modeling by PDS where the subsets of permissions are encoded into stack symbols.

This paper is organized as follows: Sect. 2 presents a summary of the IBAC syntax and behavior. In Sect. 3 we present the EWPDS definitions, and we describe the structure and the semantics of our EWPDS-based model. We show some examples of the model and we discuss the model-checking problem of our EWPDS model. Section 4 describes an implementation of our model using existing tools for EWPDS model-checking, and Sect. 5 concludes the paper.

2. IBAC Program

2.1 Syntax

We review the syntax and the semantics of an IBAC program defined in [2]. Figure 1 shows the syntax of a subset (fields and records are not taken in consideration) of commands from the cited paper. S, C, E, R, p, and x represent a sequence of commands, a command, an expression, a subset

$S ::= \epsilon \mid C; S$	command sequence
$C ::= x := E \mid p() \mid$	assignment; procedure call
$\mathbf{grant}\ R\ \mathbf{in}\ p() \mid$	assert dynamic permissions
$\mathbf{if}\ E\ \mathbf{then}\ S\ \mathbf{else}\ S \mid$	conditional
$\mathbf{test}\ R\ \mathbf{then}\ S\ \mathbf{else}\ S \mid$	check & branch on permissions
$\mathbf{test}\ R\ \mathbf{for}\ x$	check value's permissions

Fig. 1 IBAC language syntax of commands.

Manuscript received July 18, 2013.
Manuscript revised November 11, 2013.

[†]The authors are with Kochi University of Technology, Kamishi, 782–8502 Japan.
a) E-mail: 156010g@gs.kochi-tech.ac.jp
b) E-mail: takata.yoshiaki@kochi-tech.ac.jp

DOI: 10.1587/transinf.E97.D.1149

of permissions, a procedure, and a variable, respectively.

Like previous access control models (SBAC, HBAC), the user assigns a set of permissions to each procedure. We assume that the elements of this set are atomic and without any hierarchy. This set is called static permissions and are not changed during the execution. They represent the grade of authorization of the procedure, i.e. which operations that procedure can perform or not. On the other hand the runtime system dynamically maintains the current permissions, called dynamic permissions, of the execution process based on the static permissions of each procedure called. These dynamic permissions change during the execution of the program, generally at call and return statements. When a procedure is called, the current permissions of the process are intersected with the static permissions of the called procedure. When the procedure finishes the behavior of the dynamic permissions varies depending the access control model.

In SBAC the process recovers the previous current permissions when the called procedure finishes. This means that SBAC does not retain any information of the permissions of previously executed methods. On the other hand, in HBAC the current permissions are maintained when the called procedure finishes. This way, HBAC keeps the history of the authorization level of previously executed methods. In all three access control models, a permission-check command **test** R **then** S_1 **else** S_2 is statically placed by the programmer just before each security-sensitive operation. At the permission-check command it is tested whether the current dynamic permissions includes as a subset the set of permissions specified in the check command. The negative case of this check is treated as a security violation. Because SBAC restores the dynamic permissions when a procedure finishes, this check command may not detect that a procedure without enough authorization level had influence in the security-sensitive operation. This is solved by HBAC because it does not restore the dynamic permissions when a procedure finalizes. However HBAC also prevents security operations in the case when a previously executed procedure without enough permissions to execute that operation does not have any influence on that operation.

IBAC solves the high restrictiveness of HBAC by including another set of dynamic permissions for each variable in the program. This permissions of variables are updated after an assignment command $x := E$ to the intersection of the static permissions of the current program block and the permissions of all variables in E. This set of permissions given to x represents the trustfulness of the value assigned to x. The IBAC includes also another kind of permission-check command called **test** R **for** x. This command inspects whether the set of permissions assigned to the variable x contains the set of permissions R.

For the convenience of the definition, we modify the syntax of a command sequence S in Fig. 1 as $S ::= n \mid n\colon C; S$ where n is a *program point*. We also call a program point a *node*, because it corresponds to a node in a control flow graph.

Formally, an IBAC program is a 7-tuple $\pi = (PR, NO, IS, p_0, PRM, SP, VR)$ where PR is a finite set of procedures, NO is a finite set of nodes (i.e. program points), $IS : PR \to S$ is a function for defining the body of each procedure, $p_0 \in PR$ is the main procedure, PRM is a finite set of *permissions*, $SP : PR \to 2^{PRM}$ is the static assignment of permissions to procedures, and VR is a finite set of global variables. We sometimes write PR_π, NO_π, and so on to indicate that those are components of a program π.

Intuitive meanings of commands are as follows.

- $x := E$ where $x \in VR$ is the assignment command. The intersection of the permissions of all the variables included in E and also the program counter variable pc is assigned to x.
- $p()$ and **grant** R **in** $p()$ where $p \in PR$ and $R \subseteq PRM$ are the procedure call commands. The former is a special case of the latter in which $R = \emptyset$. The parameter R is called *grant permissions*.
- **if** E **then** S_1 **else** S_2 is the conditional clause.
- **test** R **then** S_1 **else** S_2 is the *test command for current permissions*, which tests whether or not the subset R of permissions is included in the current *dynamic permissions*. If $R \subseteq D$ where D is the set of current dynamic permissions, then the execution advances to S_1. On the contrary case, the program advances to S_2.
- **test** R **for** x where $x \in VR$ and $R \subseteq PRM$ is the *test command for a value's permissions*. If the permissions assigned to the variable x include R as a subset, then the execution continues. Otherwise, it is aborted.

For each procedure p, a subset $SP(p)$ of permissions is assigned statically before execution. $SP(p)$ is called the *static permissions* of p. We extend the domain of SP to NO; i.e., $SP(n) = SP(p)$ if n belongs to $IS(p)$.

We write the initial program point of a command sequence S as $head(S)$; i.e., $head(n) = n$ and $head(n\colon C; S) = n$. Similarly, the last program point of S is denoted as $last(S)$; i.e., $last(n) = n$ and $last(n\colon C; S) = last(S)$. We also define $head(p) = head(IS(p))$ and $last(p) = last(IS(p))$ for $p \in PR$. $head(p_0)$ is the starting program point of the program.

The control flow graph of a sample IBAC program is shown in Fig. 2. Each procedure is represented by the set of nodes surrounded by a rectangle. The static permissions of a procedure are attached to its rectangle. The intra-procedure control flows are denoted as dotted arrows, which we call *transfer edges*. The inter-procedure control flows are denoted as solid arrows, which we call *call edges*.

The set of variables in an expression E is denoted as $V(E)$. For a command sequence S, $write\_oracle(S) = \{ x \in VR \mid S$ contains a command $x := E \}$ is the subset of variables that are potentially altered during the execution of S.

For a given program π, we model the transition system that represents the behavior of π as an extended *weighted pushdown system* (EWPDS) [5].

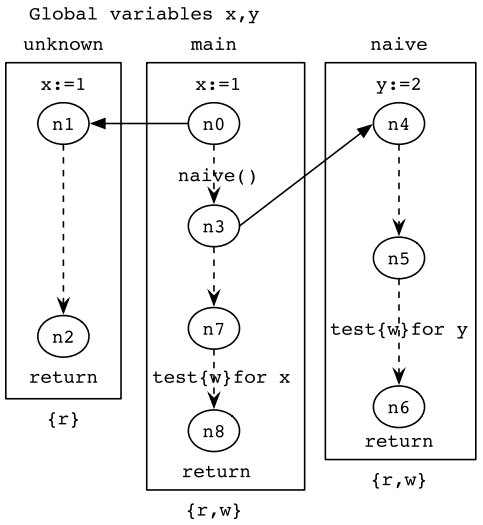

Fig. 2 A basic IBAC program.

```
GLOBAL VARIABLE x
Static Permissions of main  = {Read,Write}
Static Permissions of BadFunction = {Read}

BadFunction(){
  x = "password.txt";
  return;}

int main(){
  BadFunction();
  test {write} for x;
  write x;
  return;}

Example a

BadFunction(){
  return;}

int main(){
  x = "password.txt";
  BadFunction();
  test {write} for x;
  write x;
  return;}

Example b
```

Fig. 3 Program examples of IBAC and HBAC.

2.2 Original Semantics

In [2], the semantics of a command sequence is represented by a relation $(S, s) \Downarrow_D^P s'$ where s and s' are *stores* and P and D are subsets of permissions. A store maps each variable to a *framed* value $R[v]$ that is a pair of a subset R of permissions and a value v. The expression $(S, s) \Downarrow_D^P s'$ means that the execution of S transforms s into s' if the static permissions of S is P and the current permissions of the process is D. Similarly, $(E, s) \Downarrow_D^P R[v]$ means that the expression E is evaluated to $R[v]$ if the current store is s, the static permissions of the current procedure is P, and the current permissions of the process is D. For a store s, $s[x \mapsto E]$ denote the same store except that the value of x is E. For a procedure p, $p() = R[S]$ means that the static permissions of p is R and the body of p is S.

Variable *pc* (the program counter) is used for keeping track of implicit influence between variables caused by conditional clauses. $write\_oracle(S, s)$ represents the set of variables updated in S. If $(S, s) \Downarrow_D^P s'$ and $write\_oracle(S, s) = V$, then V is the set of variables that are potentially updated from s to s'. $taint(R, V, s)$ is a store s' such that $s'(x) = s(x)$ for $x \notin V$ and $s'(x) = s(x) \cap R$ for $x \in V$. *taint* represents an operation that reduces the current permission of variables in V. The semantics of IBAC programs is as follows:

$$\frac{p() = R[S], \quad (S, s) \Downarrow_{D \cap R}^{R} s'}{(p(), s) \Downarrow_D^P s'} \tag{1}$$

$$\frac{(S_1, s) \Downarrow_D^P s_1, \quad (S_2, s_1) \Downarrow_D^P s'}{(S_1; S_2, s) \Downarrow_D^P s'} \tag{2}$$

$$\frac{R \subseteq D, \quad (S_1, s) \Downarrow_D^P s'}{(\textbf{test } R \textbf{ then } S_1 \textbf{ else } S_2, s) \Downarrow_D^P s'} \tag{3}$$

$$\frac{R \not\subseteq D, \quad (S_2, s) \Downarrow_D^P s'}{(\textbf{test } R \textbf{ then } S_1 \textbf{ else } S_2, s) \Downarrow_D^P s'} \tag{4}$$

$$\frac{(E, s) \Downarrow_D^P P'[v], \quad R \subseteq P'}{(\textbf{test } R \textbf{ for } E, s) \Downarrow_D^P s} \tag{5}$$

$$\frac{(S, s) \Downarrow_{D \cup (R \cap P)}^{P} s'}{(\textbf{grant } R \textbf{ in } S, s) \Downarrow_D^P s'} \tag{6}$$

$$\frac{(E, s) \Downarrow_D^P R[v]}{(x := E, s) \Downarrow_D^P s[x \mapsto s(pc) \cap P \cap R[v]]} \tag{7}$$

$$\frac{\begin{array}{l}(E, s) \Downarrow_D^P R[false], \quad s_0 = s[pc \mapsto s(pc) \cap R] \\ (S_2, s_0) \Downarrow_D^P s_2, \quad V = write\_oracle(S_1, s) \\ s' = taint(s_0(pc), V, s_2)\end{array}}{(\textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2, s) \Downarrow_D^P s'[pc \mapsto s(pc)]} \tag{8}$$

$$\frac{\begin{array}{l}(E, s) \Downarrow_D^P R[true], \quad s_0 = s[pc \mapsto s(pc) \cap R] \\ (S_1, s_0) \Downarrow_D^P s_1, \quad V = write\_oracle(S_2, s) \\ s' = taint(s_0(pc), V, s_1)\end{array}}{(\textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2, s) \Downarrow_D^P s'[pc \mapsto s(pc)]} \tag{9}$$

Rules 1 and 2 define the behavior for the procedure call command and for a command sequence respectively. Rules 3 and 4 are the rules for the dynamic permission test statement, when it succeeds and when it fails respectively. Rule 5 defines the test for the permissions of a variable. In this case there is no "else" branch. Rule 6 defines the semantics for the grant operation and rule 7 is the rule for the assignment statement. Finally rules 8 and 9 are stand for the conditional clause. In these last two rules, the IBAC introduces two operations called *write_oracle* and *taint*. Basically, *write_oracle* is the set of the variables that are updated in a given command sequence, and *taint* imposes a set of permissions on a set of variables. In the conditional clause, the potentially-updated variables of the not taken branch are influenced by the variable of the branch condition. Therefore, by using the two operations mentioned above, we ensure the permissions of the branch condition variable intersect with the variables that may be updated in the not taken branch.

2.3 Examples

Example 1. Figure 3 shows a small program that illustrates

the advantage of IBAC over HBAC. In this program, the trusted function *main* calls the non-trusted function *BadFunction*, and then the function *main* tries to modify the file *password.txt*. In the version *a* of this program, the *password.txt* file is requested and returned by *BadFunction*, which does not possess any permission over the file and thus the main function would not be able to modify the file. In this example, HBAC would detect this security violation by checking the dynamic permissions just before the sensitive operation *write(x)*. Since these dynamic permissions are at most the static permissions of *BadFunction*, the operation is not executed because *BadFunction* does not have sufficient permission. IBAC also detects this violation by checking the permissions associated with variable *x* using the test command before the sensitive operation. Since this variable was modified at *BadFunction*, the operation is not executed because the permissions associated to variable *x* are the same as the static permissions of *BadFunction*. On the other hand, in the version *b* of this program, the *BadFunction* does not have any influence on the *password.txt* file. Therefore the operation *write(x)* should be allowed to be executed. However, HBAC also negates the execution of the sensitive operation because, since *BadFunction* is called, the dynamic permissions are still intersected with the dynamic permissions of *BadFunction*. As a result, the behavior of HBAC is exactly the same in this example as in example *a*. IBAC on the contrary allows the sensitive operation to be executed in example *b* because the variable *x* is modified in the function *main,* not in *BadFunction.* As a result, the permissions associated to *x* are the static permissions of *main* which are enough to execute the sensitive operation *write(x)*. Therefore, in this case the test command succeeds and the execution continues. This example illustrates how in HBAC is more restrictive than IBAC because IBAC tracks the permissions of each variable along the execution of the program, whereas HBAC just tracks the dynamic permissions of the whole execution.

Example 2. Resurrecting Duckling policy [8] is a policy such that a device is first "free" (not bound with any user) and then gets bound to the first user who tries to use the device. After this, the device is only allowed to be used by this first user until the device is "restored" to its unbound state, where any user can become the master of the device. Figure 4 shows an IBAC program that represents this policy. In this example, a global variable *x* represents a device, the functions *imprintA* and *imprintB* represent the action of binding the device to user A and B respectively, and the functions *killA* and *killB* represent the action of unbinding the device from user A and B respectively. The permissions *Pa* and *Pb* are used to determine the owner of the device represented by the variable *x*. If the variable *x* has both permissions the device is "free", i.e. it can be bound to any user. On the other hand, if *x* has only the permission *Pa* or only the permission *Pb*, the device is bound to user A or user B respectively. In order to bind the device to a user, the assignment command of both *imprint* functions intersects the set of permissions of *x* with the static permissions of the

```
int x;  //global variable

main(){                 imprintA(){
  imprintA();              test{Pa,Pb} for x;
  imprintB();              x:=1;
  killB();                 return;}
  killA();
  imprintB();           imprintB(){
  return;}                 test{Pa,Pb} for x;
                           x:=1;
killA(){                   return;}
   test{Pa} for x;
   x:=1;                killB(){
   return;}                test{Pb} for x;
                           x:=1;
                           return;}
Static permissions
main             = {Pa,Pb}
killA and killB  = {Pa,Pb}
imprintA         = {Pa}
imprintB         = {Pb}
```

Fig. 4 Program example that models the resurrecting duckling policy.

imprint function, and thus the variable *x* gets bound to user A in case of calling *imprintA* or to user B in case of calling *imprintB*. *Test* statements are placed at the beginning of both *imprint* functions in order to check if the device is not bound to any user. When a user wants to unbind the device, the function *kill* is called. This function first checks if the device is bound to the user A in case of *killA* or to the user B in case of *killB*. Then, if the *test* statement succeeds, the assignment command restores the permissions of *x* to {*Pa*, *Pb*}, which means that the device is again unbound and can be bound to any user.

In the example of Fig. 4, the *main* function first calls the function *imprintA* in order to bind the device to the user A. Then, user B tries to use the device by calling the functions *imprintB* and *killB*, but these actions are prevented by the *test* statements of those functions because the device is bound to user A. However, after user A unbinds the device by calling the function *killA*, the function *imprintB* succeeds because the device is in its unbound state.

The behavior of the Resurrecting Duckling policy can be modeled by the IBAC model as we have shown here. However it would not be possible for this policy to be represented by any HBAC model, because the set of dynamic permissions in HBAC must necessarily become smaller, and as a result the behavior of "restoring" to a previous state in which the permissions of an element are greater than before cannot be modeled using HBAC.

3. Weighted Pushdown System-Based Model

3.1 Weighted Pushdown System

DEFINITION 1. *A* **pushdown system** *is a triple* $P = (P, \Gamma, \Delta)$ *where P is the set of states or* **control locations**, Γ *is the set of* **stack symbols** *and* $\Delta \subseteq P \times \Gamma \times P \times \Gamma^*$ *is a finite set of* **transition rules**. *A* **configuration** *of P is a pair* $\langle p, w \rangle$ *where* $p \in P$ *and* $w \in \Gamma^*$. *A transition rule* $r \in \Delta$ *is written as* $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ *where p,* $p' \in P$, $\gamma \in \Gamma$ *and* $w \in \Gamma^*$. *A transition rule* $\langle p, \gamma \rangle \hookrightarrow \langle p', w \rangle$ *is called a*

push rule *if the length of w is more than one. The transition relation* $\Rightarrow$ *on configurations of P is defined as follows: If* $r{=}\langle p,\gamma\rangle \hookrightarrow \langle p',w\rangle$, *then* $\langle p,\gamma w'\rangle \Rightarrow \langle p',ww'\rangle$ *for all* $w'{\in}\,\Gamma^*$

The reflexive and transitive closure of $\Rightarrow$ *is denoted by* $\Rightarrow^*$.

For the modeling of an IBAC program, we need just one control location and thus we write $\gamma \hookrightarrow w$ instead of $\langle p,\gamma\rangle \hookrightarrow \langle p,\ w\rangle$.

DEFINITION 2. *A bounded idempotent semiring is a quintuple* $(D, \oplus, \otimes, 0, 1)$ *where* $0,\ 1 \in D$, *and*

1. $(D, \oplus)$ *is a commutative monoid with* 0 *as its unit element, and* $\oplus$ *is idempotent (i.e., for all* $a \in D$, $a \oplus a = a$*).*
2. $(D, \otimes)$ *is a monoid with the neutral element* 1.
3. $\otimes$ *distributes over* $\oplus$.
4. 0 *is annihilator with respect to* $\otimes$, *i.e., for all* $a \in D$, $a \otimes 0 = 0 \otimes a = 0$.
5. *There are no infinite descending chains for the partial order* $\sqsubseteq$ *defined as follows:* $\forall a, b \in D$, $a \sqsubseteq b$ *iff* $a \oplus b = a$.

DEFINITION 3. *A* **weighted pushdown system** *is a triple* $W = (P, S, w)$, *where* $P = (P, \Gamma, \Delta)$ *is a pushdown system,* $S = (D, \oplus, \otimes, 0, 1)$ *is a bounded idempotent semiring, and* $w : \Delta \to D$ *is a function that assigns a value from D to each rule of P.*

The extend operation $\otimes$ is used for computing a weight of a single path, while the combine operation $\oplus$ is used for combining the weights of joining paths. To a rule sequence $\sigma{=}r_1 r_2 \ldots r_k$, a weight $v(\sigma) {=} w(r_1){\otimes}w(r_2){\otimes}\cdots \otimes w(r_k)$ is associated by the EWPDS. For configurations s and t, let $path(s, t)$ be the set of all rule sequences that transform s into t. The *meet-over-all-valid-paths* value MOVP(s, t) is defined as $\oplus\{v(\sigma)|\sigma \in path(s,t)\}$.

In the EWPDS W_π that models an IBAC program π, the stack alphabet Γ is the set *NO* of nodes and the configuration of the PDS part of W_π is a finite sequence of nodes that represents the call stack. The dynamic assignment of permissions to variables is codified on the weights of the EWPDS as explained as follows.

DEFINITION 4. *If G is a finite set, then the* **relational weight domain** *on G is defined as a the bounded idempotent semiring* $(2^{G\times G},\ \cup, ;,\ \emptyset,\ id)$ *where weights are binary relations on G, combine is union, extend is relational composition,* 0 *is the empty relation, and* 1 *is the identity relation id on G.*

We define $VR' = VR \cup \{pc, dp\}$ where pc and dp are newly introduced variables for representing the set of current dynamic permissions of the program counter and the execution process, respectively. An *environment* is an assignment of permissions to variables in VR' and is a function from VR' to 2^{PRM}. The set of all environments is denoted as Env. In our model, we use the relational weight domain on Env. Therefore, a weight of a EWPDS W_π is of the form:

$$w = \{(e, e') \mid e, e' \in Env, \ldots\}.$$

A weight is a set of pairs and the first component of each pair represents the pre-state of the variables before applying the transition rule. The second component represents the post-state of the variables after applying the transition rule.

For an environment e, $e\,[x \mapsto R]$ denote the same environment except that the value of x is R.

When a conditional clause finishes, the variable pc needs to be restored to its value before the conditional clause. The same issue occurs with the variable dp in case of a procedure call. In order to implement this behavior, we use the Extended-WPDS (EWPDS) [5], which allows local variables to be stored at call sites and then, when a procedure finishes, combine the returned value with the stored value by using a merging function. For a semiring S on domain D, a *merging function* is defined as follows:

DEFINITION 5. *A function* $g : D \times D \to D$ *is a* **merging function** *with respect to a bounded idempotent semiring* $S = (D, \oplus, \otimes, 0, 1)$ *if it satisfies the following properties.*

1. **Strictness**. *For all* $a \in D$, $g(0, a) = g(a, 0) = 0$.
2. **Distributivity**. *The function distributes over* $\oplus$.
3. **Path Extension**. *For all* $a, b, c \in D$, $g(a \otimes b, c) = a \otimes g(b, c)$.

DEFINITION 6. *An* **extended weighted pushdown system** *is a quadruple We = (P, S, w, g) where (P, S, w) is a weighted pushdown system and* $g : \Delta_2 \to G$ *assigns a merging function to each rule in* Δ_2 *, where G is the set of all merging functions on the semiring S and* Δ_2 *is the set of push rules of P.*

Using the merging functions of the EWPDS at the end of a conditional clause and at the end of a procedure call, the values of pc and dp are restored respectively. Regarding the rest of the variables in the weight, they remain unaffected by the merging function. Assuming w_1 to be the weight just before a conditional clause or a procedure call and w_2 to be the weight after a conditional clause or a procedure call, the merging functions are defined as follows:

- For a conditional clause:

$$g_1(w_1, w_2) = \{(e, e_2\,[pc \mapsto e_1(pc)]) \Big| \\ e, e_1, e_2 \in Env,\ (e, e_1) \in w_1, (e_1, e_2) \in w_2\}$$

- For a procedure call:

$$g_2(w_1, w_2) = \{(e, e_2\,[dp \mapsto e_1(dp)]) \Big| \\ e, e_1, e_2 \in Env,\ (e, e_1) \in w_1, (e_1, e_2) \in w_2\}$$

In case of conditional clause, the variable pc is restored to its value in the weight w_1, the one before the conditional clause. The rest of variables are set to their value in weight w_2. In case of procedure call, the same process is performed but restoring the variable dp instead. For other push rules we assign the third merging function $g_0(w_1, w_2) = w_1 \otimes w_2$, which is the same as the combining operation.

3.2 Model Semantics

The set $\Delta(\pi)$ of transition rules of W_π is defined as $\Delta(\pi) = \bigcup_{p \in PR_\pi} \Delta(IS_\pi(p))$, and $\Delta(S)$ for a command sequence S is defined as the least set that satisfies the following inference rules. Moreover, the weight specified in each inference rule is assigned to the transition rule defined in that rule.

$$\frac{n' = head(S)}{\Delta(n\colon C; S) = \Delta(n\colon C; n') \cup \Delta(S)} \tag{10}$$

$$\begin{array}{l} t = n \hookrightarrow n' \in \Delta(n\colon x := E; n') \\ \quad w(t) = \{(e, e[x \mapsto P]) \mid e \in Env \\ P = (\bigcap_{y \in V(E)} e(y)) \cap SP(n) \cap e(pc)\} \end{array} \tag{11}$$

$$\frac{m = head(p)}{\begin{array}{l} t = n \hookrightarrow m\, n' \in \Delta(n\colon \textbf{grant } R \textbf{ in } p(); n') \\ \quad w(t) = \{(e, e[dp \mapsto D]) \mid e \in Env \\ D = (e(dp) \cup R) \cap SP(p)\} \quad g(t) = g_2 \end{array}} \tag{12}$$

$$\frac{p \in PR, \quad m = last(p)}{t = m \hookrightarrow \epsilon \in \Delta(m) \qquad w(t) = id} \tag{13}$$

$$\begin{array}{l} t = n \hookrightarrow n' \in \Delta(n\colon \textbf{test } R \textbf{ for } x; n') \\ \quad w(t) = \{(e, e) \mid e \in Env,\ R \subseteq e(x)\} \end{array} \tag{14}$$

$$\begin{array}{l} i, j \in \{1, 2\}, \quad i \neq j, \quad m = head(S_i) \\ m' = last(S_i), \quad W = \text{write\_oracle}(S_j) \\ \hline t_1 = n \hookrightarrow m\, n' \in \Delta(n\colon \textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2; n') \\ \quad w(t_1) = \{(e, e[pc \mapsto P]) \mid e \in Env, \\ \qquad P = e(pc) \cap SP(n) \cap (\bigcap_{y \in V(E)} e(y))\} \\ \qquad g(t_1) = g_1 \\ t_2 = m' \hookrightarrow \epsilon \in \Delta(m') \\ \quad w(t_2) = \{(e, e[x \mapsto e(x) \cap e(pc) \mid x \in W]) \\ \mid e \in Env\} \\ \Delta(n\colon \textbf{if } E \textbf{ then } S_1 \textbf{ else } S_2; n') \supseteq \Delta(S_1) \cup \Delta(S_2) \end{array} \tag{15}$$

$$\begin{array}{l} m = head(S_1), \quad m' = last(S_1) \\ \hline t_1 = n \hookrightarrow m\, n' \in \Delta(n\colon \textbf{test } R \textbf{ then } S_1 \textbf{ else } S_2; n') \\ \quad w(t_1) = \{(e, e) \mid e \in Env,\ R \subseteq e(dp)\} \\ \qquad g(t_1) = g_0 \\ t_2 = m' \hookrightarrow \epsilon \in \Delta(m') \qquad w(t_2) = id \\ \Delta(n\colon \textbf{test } R \textbf{ then } S_1 \textbf{ else } S_2; n') \supseteq \Delta(S_1) \cup \Delta(S_2) \end{array} \tag{16}$$

$$\begin{array}{l} m = head(S_2), \quad m' = last(S_2) \\ \hline t_1 = n \hookrightarrow m\, n' \in \Delta(n\colon \textbf{test } R \textbf{ then } S_1 \textbf{ else } S_2; n') \\ \quad w(t_1) = \{(e, e) \mid e \in Env,\ R \not\subseteq e(dp)\} \\ \qquad g(t_1) = g_0 \\ t_2 = m' \hookrightarrow \epsilon \in \Delta(m') \qquad w(t_2) = id \\ \Delta(n\colon \textbf{test } R \textbf{ then } S_1 \textbf{ else } S_2; n') \supseteq \Delta(S_1) \cup \Delta(S_2) \end{array} \tag{17}$$

Rule (10) defines the set of transition rules for a command sequence.

Rule (11) is the rule for the assignment command. If the control reaches an assignment node n, then the next current node can be the node n' next to n. The weight of the rule states that the permissions of the variable x is intersected with three sets of permissions: the permissions of all the variables in the expression E, the static permissions of the current node, and the permissions of the program counter.

Rule (12) states that if the control is at a node n that is a call to a procedure p, then the initial node m of p can be pushed onto the stack. In the weight of the rule, the dynamic permissions are updated to $D = (D' \cup R) \cap SP(n)$ where D' is the old value of dp.

Rule (13) describes the return from a procedure. If the current node m is the last node of a procedure p, then m is simply removed from the stack and the next current node is the node n' next to the caller node, which is placed int he stack by Rule (12). Regarding to the weight, the value of the dynamic permissions is restored to the one before the procedure call by the merging function g_2.

Rule (15) describes the behavior when the control reaches a conditional clause. If the current node n is a conditional clause, then the next current node can be the initial node of either the *then* clause S_1 or the *else* clause S_2. The EWPDS takes non-deterministically one of the two branches. If the control reaches the last node m' of S_1 or S_2, then m' is simply removed from the stack and the next current node is the node n' next to n. There are two changes regarding the weight of these rules. First, the permissions of the program counter pc are intersected with the permissions of all the variables included in the expression E. Second, at the end of the conditional clause, the permissions of pc is imposed to the variable x that are updated in the not taken branch. A push rule is needed for the conditional clause because, in case of nested *if* commands, the pc variable has to be tracked accordingly.

Rules (16) and (17) model the behavior of SBAC *checkPermission* statement. In these rules, when the control reaches a test node n, the next current node can be the initial node of either the *then* clause S_1 or the *else* clause S_2. Regarding to the weight of these rules, advancing to S_1 is valid only when $R \subseteq D$ and advancing to S_2 is valid only when $R \not\subseteq D$ where D is the current dynamic permissions.

Finally, Rule (14) says that if control reaches a test node n for a variable x, and the current dynamic permissions of x include R, then the next current node can be the node n' next to n. In this case, the weight keeps the environment as the same. If the permissions of x does not include R, then the weight does not map the environment to any environment.

3.3 Model Example

Example 3. Let us return to the IBAC program π_1 in Fig. 2. When the unknown procedure is called by n_0, the current dynamic permissions i.e., the variable dp in the weight of the EWPDS, become $e(dp) \cap SP(n_1) = e(dp) \cap \{r\}$ where e is an initial environment. In node n_1, because the variable x is updated, the permissions associated to variable x become $e(pc) \cap SP(n_1) = e(pc) \cap \{r\}$. Therefore, the test at node n_7 fails regardless of the initial environment because the permissions of x do not include $\{w\}$. However, the test at node n_5 succeeds if $w \in e(pc)$ because the variable y is just modified at the naive procedure, which has

Transitions	Weight of the path from n0
$n_0 \Rightarrow n_1 n_3$	$\{(e, e\,[dp \mapsto \{r\} \cap e\,(dp)])\}$
$\Rightarrow n_2 n_3$	$\{(e, e[dp \mapsto \{r\} \cap e\,(dp)\,,$ $x \mapsto \{r\} \cap e\,(pc)])\}$
$\Rightarrow n_3$	$\{(e, e\,[x \mapsto \{r\} \cap e\,(pc)])\}$
$\Rightarrow n_4\ n_7$	$\{(e, e[dp \mapsto \{r, w\} \cap e\,(dp)\,,$ $x \mapsto \{r\} \cap e\,(pc)\,])\}$
$\Rightarrow n_5\ n_7$	$\{(e, e[dp \mapsto \{r, w\} \cap e(dp),$ $x \mapsto \{r\} \cap e(pc), y \mapsto \{r, w\} \cap e(pc)])\}$
$\Rightarrow n_6\ n_7$	$\{(e, e[dp \mapsto \{r, w\} \cap e(dp),$ $x \mapsto \{r\} \cap e(pc), y \mapsto \{r, w\} \cap e(pc)])\}$
$\Rightarrow n_7$	$\{(e, e[x \mapsto \{r\} \cap e\,(pc)\,,$ $y \mapsto \{r, w\} \cap e\,(pc)\,])\}$
$\Rightarrow n_8$	$\{\,\}$
$\Rightarrow \varepsilon$	$\{\,\}$

Fig. 5 Transitions of $W_{\pi 1}$.

the permissions $e(pc) \cap \{r, w\}$. This test at node n_5 would have failed in an HBAC program because the first call to the unknown method would have cut the permission $\{w\}$ for the rest of the execution, even if y is not modified in the unknown method. The transitions of the EWPDS $W_{\pi 1}$ and the weight after each transition following the semantics explained before are shown in Fig. 5.

In order to restore the *dp* variable at the finalization of a procedure, the merging function g_1 is applied at the rules for the end of a procedure, in this case the third and the seventh transitions. In the last transition of this example, the weight becomes the empty relation because the test statement at node n_7 fails.

3.4 Formal Verification Problem

Let us discuss in this section the model-checking problem of our EWPDS model. Let π be an IBAC program and W_π be the EWPDS that models that program. The initial environment is an environment e_0 such that $e_0(pc) = PRM$ and $e_0(dp) = SP(p_0)$. We consider the reachability problem on π, i.e., check whether or not a given node n is reachable from the initial configuration $n_0 = head(p_0)$ with the initial environment e_0. The property that an invalid node n is not reachable from the initial program node can be represented as the following expression on W_π:

$$(e_0, e) \notin \mathrm{MOVP}\,(n_0, n\xi) \text{ for any } e \text{ and } \xi.$$

This expression means that when the EWPDS reaches some configuration whose stack top is n, the weight in that configuration must not map the initial environment to any environment. Otherwise the expression does not hold and the IBAC program is invalid because it successfully reaches an invalid configuration. As an example, let us take the program π_2 in Fig. 2. Following the original semantics of the IBAC model, in program π_2 the node n_8 is not reachable because the test command at n_7 should abort the execution. To verify that this behavior occurs in $W_{\pi 2}$, the above expression for n_8 is examined. That expression holds because the weight at node n_8 does not map e_0 to any environment regardless the path chosen at the conditional clause. Therefore, the safety property of the original program π_2 is maintained in the EWPDS $W_{\pi 2}$.

Let us consider more complex verification problem. For a program like the Resurrecting Duckling policy in Example 2, one may want to verify whether a given bad path does not exist in that program. Let $n_1 = last(imprintA)$, $n_2 = last(killA)$, and $n_3 = last(imprintB)$. Then one of the bad paths is $n_0 \Rightarrow^* n_1\xi \Rightarrow^* n_3\xi'$ for some ξ, ξ' such that its second half $n_1\xi \Rightarrow^* n_3\xi'$ does not contain $n_2\xi''$ for any ξ''. Using a technique for model-checking PDS [6], we can obtain a EWPDS W' from W_π such that the transition relation $\Rightarrow$ for W' is a subset of that of W_π and, in W', the stack top must transit according to a regular expression $n_0\, NO^*\, n_1\, (NO - \{n_2\})^*\, n_3$. Conducting the unreachability test for W', we can verify whether the bad path does not exist in program π.

A practical example of the usefulness of this method would be the detection of bugs in a given IBAC program. Following the Example 2, imagine there is a bug in the test command of the *killB()* function so instead of *testPb for x* the programmer wrote *testPa for x*. In this case, when the *killB()* function is called it succeeds and the user B would be able to operate the device even though the device belongs to user A. This error could be detected by using the MOVP from the starting point of function imprintA until the return of function killB. In this example, the MOVP is not empty because there is a path that reaches the end of killB from imprintA, which is a security violation. Therefore, by checking the value of the MOVP we can detect these type of bugs in an IBAC program.

The soundness of our EWPDS model with respect to the original semantics of IBAC programs given in [2] is represented by the following Theorem 1. Note that the original semantics is defined with respect to *stores*, which map each variable $x \in VR \cup \{pc\}$ to a *framed value* $R[v]$, i.e. a pair of permissions R and a value v. The environments we used abstract the values and consider only the permissions. We define a projection function *proj* over framed values as $proj(R[v]) = R$. Moreover, for a store s and a subset D of permissions, we define $proj(s, D)$ as the environment e such that $e(dp) = D$ and $e(x) = proj(s(x))$ for $x \in VR \cup \{pc\}$. We define $SP(S) = SP(head(S))$ for a command sequence S.

Theorem 1 (Soundness)**.** *Given an IBAC program π, if $(S, s) \Downarrow_D^{SP(S)} s'$ for a command sequence S in π and stores s and s' and dynamic permissions $D \subseteq PRM$, then the EWPDS W_π satisfies $n_0 \Rightarrow^* n_1$ and $(e, e') \in \mathrm{MOVP}(n_0, n_1)$ where $n_0 = head(S)$, $n_1 = last(S)$, $e = proj(s, D)$, and $e' = proj(s', D)$.*

Proof. This theorem can be proved by induction on the number l of steps to derive $(S, s) \Downarrow_D^{SP(S)} s'$ (Shown in Ap-

pendix A). □

The above theorem says that the non-reachable states of a transition system W_π are neither reachable in the IBAC program π. However, due to the non-deterministic behavior of the EWPDS, if W_π includes conditional clauses, its reachable set of states is greater than the program π.

4. Implementation and Performance Evaluation

The model presented in this paper is implemented using the model-checker for EWPDS called WALi [9]. This tool provides a C++ interface for easily creating and verifying EWPDS and also provides an add-on that implements a binary relation domain using the Binary Decision Diagram (BDD) library Buddy [10]. The implementation of a binary relation includes the basic semiring methods that WALi needs in any weight domain. These methods are: *One()* and *Zero()* which return the neutral element 1 and the empty element 0 of the semi-ring respectively, *Combine()* which returns the union of two semiring elements, and *Extend()* which returns the composition of two semiring elements.

Besides the previous top-level methods of the relational weight domain, additional low-level methods are implemented in order to create and return a semi-ring element according to the changes denoted in the semantics of our model. For example, for the assignment command, a method returns an environment that reflects the update of the set of permissions of the updated variable, and then this returned semi-ring element is passed to WALi as the weight of the corresponding EWPDS rule. The weights returned by these methods are used in the model checking computation by the Combine and Extend operations explained before. Moreover, we also implemented the two merge functions defined in Sect. 3.1.

In our model, a relational weight domain is composed by a binary relation R over D where D is a set of Boolean vectors. These vectors store the permissions associated to each global variable plus the dynamic permissions DP and the program counter variable PC. Therefore the length of each vector is $|VR'| \times |PRM|$ where $|VR'|$ is the number of variables including the two special variables PC and DP, and $|PRM|$ is the number of different permissions. For example, a vector in a program with one global variable x and two different permissions P_a and P_b would be of the form (x_{Pa}, x_{Pb}, pc_{P_a}, pc_{P_b}, dp_{P_a}, dp_{P_b}) where all the components are Boolean. The relational weight of our problem would be composed by a set of pairs of these Boolean vectors, where the two vectors of a pair would be the *pre* state and the *post* state of a weight. The ordering of the Boolean variables in BDDs chosen for these two vectors is $(x_{pre}, x_{post}, \ldots, z_{pre}, z_{post})$, instead of $(x_{pre}, \ldots, z_{pre}, x_{post}, \ldots, z_{post})$. The reason of this choice is that in the former ordering, the number of nodes of the BDD that represents our *id* weight grows linearly. On the other hand, using the latter ordering, the number of nodes grows exponentially.

Using all the elements described above, we implemented a EWPDS of four IBAC programs, each one representing a group of IBAC programs. These four programs try to include all the IBAC operations that can be used in any IBAC. This way, if our model is suitable for all these programs separately, it would be also suitable for verifying the majority of IBAC programs. The scalability of all the programs depends on the number of permissions which is augmented from 2 to 20.

The first program used in this experiment is the one shown in Fig. 4. This is chosen because it is a typical example of a security policy that cannot be modeled using previous access control models but can be modeled using IBAC. In terms of the program, increasing the number of permissions means an increment of the number of procedures, especifically 2 procedures (imprintX and killX) are added for each permission incremented. In terms of the security policy, an increment of one permission means, for example, allowing one more user the possibility to inprint a device. Therefore incrementing the number of permissions increments also the number of users that can get a device.

The rest three programs are artifitially created in order to test the rest of operations an IBAC program can use, without any intention of representing a real life situation. The first one shown in Fig. 6 tests the conditinal clause behavior in an IBAC program. The objective in this program is to check if the variable *y*, which every time that enters a conditional clause loses one permission, has the same set of permissions at the end of the conditional clause no matter the path taken. For example, if we take the first conditional clause, the variable *y* loses the permission A explicitally in the *then* path due to the assignment operation because variable *y* gets the permissions of variable *x*. If the *else* path is taken, the variable *y* also will lose the permission A at the end of the conditional clause due to the *write_oracle* and *taint* of the IBAC semantics. This fact is tested using **test** *A* **for** *y* operation which no matter the path taken should fail. This behaviour is tested in all the nested conditional clauses. The depth of the nesting increases with the number of permissions.

The second artifitial program shown in Fig. 7 is created to test the grant operation and the dynamic permission check operation. At first, the main function calls either GrtA or GrtB granting a permission that is not including in the static permissions of these functions, A in case of GrtA and B in case of GrtB. Then we check if that permission has been granted into the dynamic permissions using the **test** R **then else** command, where R is either permission A or permission B depending on the function. These test operations should always succeed in this example. After this, the permissions of variable *x* are updated to the static permissions of the function called and then checked in the main function using the **test** *A,B* **for** *x* command. This test is placed to check that the *then* path has been taken in the dynamic permission test command and thus, the permissions of the variable *x* were correctly updated. In this program, by augmenting the permissions the number of Grt() functions in-

```
Starting permissions of x = {B,C}
Starting permissions of y = {A,B,C}
SP of main = {B,C}, SP funcB= {C}
SP of funcC = {}
void main(){
  if(x){
    y=1;
    funcB();
    if(x){
      y=1;
      funcC();
    }else{}
    Test {B} for y;
  }else{}
  Test {A} for y;
  return;}

int funcB(){
  x=1;
  return;}

int funcC(){
  x=1;
  return;}
```

Fig. 6 Program test of conditional clause with 3 permissions.

```
Starting permissions of x = {A,B}
SP of main = {A,B}
SP of GrtA= {B}, SP of GrtB = {A}

      void main(){
        grant({A},GrtA())
      or
        grant({B},GrtB());
        test {A,B} for x;
        return;}

int GrtA(){        int GrtB(){
  test{A}            test{B}
  then x=1;          then x=1;
  else return;}      else return;}
```

Fig. 7 Program test of grant and dynamic permission check with 2 permissions.

```
Starting permissions of x = {A,B}
SP of main = {A,B}
SP of FuncA= {A}, SP of FuncB = {B}

void main(){            int FuncA(){
  FuncA()                 x=1;
  or                      return;}
  FuncB();
  test {A} for x
  or                    int FuncB(){
  test {B} for x;         x=1;
  main() or return;       return;}
  return;}
```

Fig. 8 Program test of loops using recursion with 2 permissions.

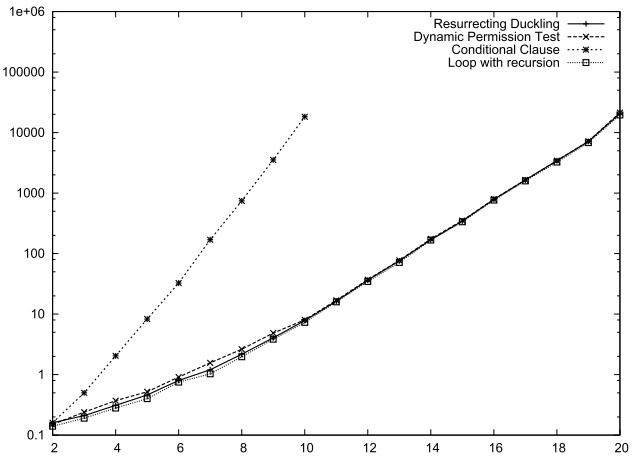

Fig. 9 Performance evaluation of a query in our EWPDS model.

creases.

Finally the third artifitial program shown in Fig. 8 tests the behavior of recursive loops. This simple program nondeterministically calls either FuncA() or FuncB() to eliminate a permission from the set of permissions of variable *x*. Then it tests also nondeterministically if x has the permission A or B. Finally the program can recursively call the main function again or finish. Note that because our model cannot explicitaly control a conditional statement, the decision of finishing or continuing the loop is taken nondeterministically by the model checker. As the number the permissions is increased, the number of functions and the number of test statements that can be executed increases.

In order to measure the performance of the implemented EWPDS, a poststar query [4] is calculated for every example. Given a set of configurations C, a poststar query calculates the set of configurations $post^*(C) := \{c' \mid \exists c \in C : c \Rightarrow^* c'\}$, i.e, the set of configurations that are reachable from elements of C via the transition relation. We performed the poststar query with the beginning of the program as the starting configuration.

The performance results are shown in Fig. 9. The x and y axes represent the number of permissions and the execution time respectively. The environment used is a Intel(R) Core(TM) i7-3770 CPU 3.40Ghz with 8 GB of RAM. As we see in this figure, all the examples except the conditional clause one the time complexity of the query is efficient until 20 permissions but from 21 permissions the time grows exponentially and becomes unreasonable for more than 20 permissions. In a real life example using the example in Fig. 4 where the number of permissions represents the number of users that can use a device, 20 permissions could be sufficient for example for a remote control in a family house, where 20 users at the same time at max seems reasonable. However, systems that require a large number of users operate simultaneously, 20 permissions would not be sufficient.

The reason for this limitation is related to the bdd library cache space. Because the number of nodes of the bdd's becomes too high, the library does not operate efficiently for more than 20 permissions. This is because the cache used by the bdd library becomes full after 20 permissions and therefore the execution time increases. In the conditional clause example, we use 2 variables instead of 1 as in the other examples. Therefore, the bdds grow faster and the cache of the library becomes full at 10 permissions. Possible optimization paths may include the redefinition of the weight codification in order to use fewer BDD nodes, and the parallelization of the BDD library [11].

5. Related Work

The verification problem for HBAC has been discussed in

works such as [3] and [12], but no formal verification has been proposed for IBAC. Our work is the first to discuss and implement the model-checking problem for IBAC programs.

PDS-based approaches are recently being used to resolve security related problems. In [14], EWPDS model-checking is used to develop a distributed certificate-chain-discovery algorithm for a trust management system called SPKI/SDSI. In [13] a PDS-based approach is utilized to develop a Symbolic PDS from core-language program in order to express noninterference with LTL formula. While they used PDS for analyzing information flow of usual procedural programs, we aim to analyze the execution paths of the IBAC programs. In IBAC, permissions are dynamically maintained at each procedure call and there exists a dynamic permission test statement; we aim to model such behavior using EWPDS. On the other hand, since our purpose is to verify the execution paths of an IBAC program, we have not considered the self-composition technique taken in [13] as a key technique for precise non-interference analysis.

In [2], Pistoia et al. proposed two types of implementation for the IBAC model: a static and a dynamic enforcement. Our method mainly aims to provide a verification method for an IBAC program on the dynamic implementation. The runtime system of the dynamic implementation maintains the subset of permissions of each value, and at the test R for x command it checks whether R is included by the subset of permissions attached to x. Our verification method can analyze the behavior of a given IBAC program on the dynamic implementation, and it can be used for verifying whether the given IBAC program is correctly written and has no unintentional behavior. In the static implementation, it is inspected before runtime whether all statements in the static backward slice of each test command have enough static permissions. If not, then the program is not executed at all. Even for the static implementation, our verification method can be used for analyzing a path that is safe in theory but is not permitted in the static implementation.

6. Conclusion and Future Work

In this paper we present a EWPDS-based formal model for dynamic access control based on information flow (IBAC). A subset of the original IBAC semantics is represented by a EWPDS. The verification problem of our model and an implementation in an existing EWPDS tool are also discussed. Theorem 1 proves the soundness of our model. However, because the values of the variables are not stored in our model, the conditional clauses are treated non-deterministically. Therefore, our model is an over-approximation of the original IBAC model and thus is only suitable for safety properties. The implementation described in Sect. 4 achieves good scalability up to 20 permissions using one variable. Future work includes the optimization of our implementation and searching for a data structure more suitable for our EWPDS model.

References

[1] M. Abadi and C. Fournet, "Access control based on execution history," Proc. 11th Network and Distributed System Security Symposium (NDSS 2003), Feb. 2003.
[2] M. Pistoia, A. Banerjee, and D.A. Naumann, "Beyond stack inspection: A unified access-control and information-flow security model," Security and Privacy, 2007. SP '07. IEEE Symposium on, pp.149–163, May 2007.
[3] J. Wang, Y. Takata, and H. Seki, "HBAC: A model for history-based access control and its model checking," 11th ESORICS, LNCS, vol.4189, pp.263–278, 2006.
[4] T. Reps, S. Schwoon, S. Jha, and D. Melski, "Weighted pushdown systems and their application to interprocedural dataflow analysis," Sci. Comput. Program., vol.58, no.1-2, pp.206–263, 2005.
[5] A. Lal, T. Reps, and G. Balakrishnan, "Extended weighted pushdown systems," CAV05: Proceedings of the 17th International Conference on Computer Aided Verification, pp.434–448, 2005.
[6] S. Schwoon, "Model-checking pushdown systems," PhD thesis, Technical University of Munich, 2002.
[7] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers, "Going beyond the sandbox: An overview of the new security architecture in the Java^{TM} Development Kit 1.2," USENIX Symp. Internet Technologies and Systems, pp.103–112, 1997.
[8] F. Stajano and R. Anderson, "The resurrecting duckling: Security issues in ad-hoc wireless networks," Security Protocols, 7th International Workshop Proceedings, LNCS, vol.1796, 1999. URL http://www.cl.cam.ac.uk/ fms27/duckling/
[9] N. Kidd, T. Reps, and A. Lal, "WALi: A C++ library for weighted pushdown systems," http://www.cs.wisc.edu/wpis/wpds/download.php, 2008.
[10] J. Lind-Nielsen, "BuDDy – A binary decision diagram package," http://sourceforge.net/projects/buddy, 2004.
[11] Y. He, Multicore-enabling a Binary Decision Diagram algorithm, http://software.intel.com/en-us/articles/multicore-enabling-a-binary-decision-diagram-algorithm, 2009.
[12] A. Banerjee and D.A. Naumann, "History-based access control and secure information flow," CASSIS04, LNCS, vol.3362, pp.27–48, 2004.
[13] C. Sun, L. Tang, and Z. Chen, "Secure information flow by model checking pushdown system," Proceedings of UIC-ATC09, pp.586–591, 2009.
[14] S. Jha, S. Schwoon, H. Wang, and T. Reps, "Weighted pushdown systems and trust-management systems," TACAS06, LNCS, vol.3920, pp.1–26, Springer-Verlag, 2006.

Appendix: Proof of Theorem 1

Proof. This theorem can be proved by induction on the number l of steps to derive $(S, s) \Downarrow_D^{SP(S)} s'$.

(Basis) Assume that $l = 0$. This implies that $s = s'$ and S contains no command, and thus $n_0 = n_1$. Since MOVP (n_0, n_0) equals the identity relation id, $(e, e) \in \mathrm{MOVP}(n_0, n_1)$ for $e = proj(s, D) = proj(s', D)$.

(Induction step) Assume that $l > 0$. This implies that $S = n_0 : C; S'$. By the definition of $\Downarrow$, $(C, s) \Downarrow_D^{SP(S)} s''$ and $(S', s'') \Downarrow_D^{SP(S)} s'$ for some s''. By the induction hypothesis, $n_2 \Rightarrow^* n_1$ and $(e'', e') \in \mathrm{MOVP}(n_2, n_1)$ where $n_2 = head(S')$, $n_1 = last(S') = last(S)$, $e'' = proj(s'', D)$, and $e' = proj(s', D)$. On the other hand, for each form of

C we can show that $n_0 \Rightarrow^* n_2$ and $(e, e'') \in g_1(id, \mathrm{MOVP}(n_0, n_2))$ for the conditional clause, $(e, e'') \in g_2(id, \mathrm{MOVP}(n_0, n_2))$ for the procedure call, or $(e, e'') \in \mathrm{MOVP}(n_0, n_2)$ for the other commands. We conclude that $n_0 \Rightarrow^* n_1$ and $(e, e') \in \mathrm{MOVP}(n_0, n_1)$.

(A) If $C = x := E$, then by the definition of $\Downarrow$, $s'' = s[x \mapsto (s(pc) \cap SP(n_0) \cap R)[v]]$ for some value v where $R = SP(n_0) \cap (\bigcap_{y \in V(E)} proj(s(y)))$. On the other hand, by the definition of W_π, $n_0 \Rightarrow n_2$ and $(e, e_2) \in \mathrm{MOVP}(n_0, n_2)$ for any $e \in Env$ and $e_2 = e[x \mapsto (\bigcap_{y \in V(E)} e(y)) \cap SP(n_0) \cap e(pc)]$. Therefore $(e, e'') \in \mathrm{MOVP}(n_0, n_2)$ when $e = proj(s, D)$ and $e'' = proj(s'', D)$. Since $\mathrm{MOVP}(n_0, n_1) \supseteq \mathrm{MOVP}(n_0, n_2) \otimes \mathrm{MOVP}(n_2, n_1)$, $n_0 \Rightarrow^* n_1$ and $(e, e') \in \mathrm{MOVP}(n_0, n_1)$ where $e = proj(s, D)$ and $e' = proj(s', D)$.

(B) If $C =$ **grant** R **in** $p()$, then by the definition of $\Downarrow$, $(IS(p), s) \Downarrow_{D'}^{SP(p)} s''$ where $D' = (D \cup R) \cap SP(p)$. By the induction hypothesis, $n_3 \Rightarrow^* n_4$ and $(e_3, e_4) \in \mathrm{MOVP}(n_3, n_4)$ where $n_3 = head(p)$, $n_4 = last(p)$, $e_3 = proj(s, D')$, and $e_4 = proj(s'', D')$. On the other hand, by the definition of W_π, $n_0 \Rightarrow n_3\, n_2$ and $(e, e_3) \in \mathrm{MOVP}(n_0, n_3 n_2)$ for any $e_0 \in Env$ and $e_3 = e[dp \mapsto (e(dp) \cup R) \cap SP(p)]$. Therefore $(e, e_3) \in \mathrm{MOVP}(n_0, n_3 n_2)$ when $e = proj(s, D)$ and $e_3 = proj(s, D')$. Moreover, $n_3\, n_2 \Rightarrow^* n_4\, n_2 \Rightarrow n_2$ and $\mathrm{MOVP}(n_3 n_2, n_2) = \mathrm{MOVP}(n_3 n_2, n_4 n_2) = \mathrm{MOVP}(n_3, n_4)$, and thus $(e, e_4) \in \mathrm{MOVP}(n_0, n_2) = \mathrm{MOVP}(n_0, n_3 n_2) \otimes \mathrm{MOVP}(n_3 n_2, n_2)$ where $e = proj(s, D)$ and $e_4 = proj(s'', D')$. By the definition of W_π, $\mathrm{MOVP}(n_0, n_1) = g_2(id, \mathrm{MOVP}(n_0, n_2)) \otimes \mathrm{MOVP}(n_2, n_1)$ and $(e, e'') \in g_2(id, \mathrm{MOVP}(n_0, n_2))$ where $e'' = proj(s'', D)$ since g_2 forces $e''(dp) = e(dp)$. It concludes that $n_0 \Rightarrow^* n_1$ and $(e, e') \in \mathrm{MOVP}(n_0, n_1)$.

(C) Consider the case that $C =$ **if** E **then** S_1 **else** S_2. Because of the symmetricalness of the definition, we assume that the value of E under s is true without loss of generality. By the definition of $\Downarrow$, $(S_1, s_0) \Downarrow_D^{SP(n_0)} s_1$ and $s'' = s_2[pc \mapsto s(pc)]$ where $s_0 = s[pc \mapsto s(pc) \cap R]$, $s_2 = s_1[x \mapsto (s(pc) \cap R \cap P)[v] \mid x \in W,\ s_1(x) = P[v]]$, $R = SP(n_0) \cap (\bigcap_{y \in V(E)} proj(s(y)))$, and $W = \mathrm{write\_oracle}(S_2)$. By the inductive hypothesis, $n_3 \Rightarrow^* n_4$ and $(e_3, e_4) \in \mathrm{MOVP}(n_3, n_4)$ where $n_3 = head(S_1)$, $n_4 = last(S_1)$, $e_3 = proj(s_0, D)$, and $e_4 = proj(s_1, D)$. By the definition of W_π, $n_0 \Rightarrow n_3\, n_2$ and $(e, e_3) \in \mathrm{MOVP}(n_0, n_3 n_2)$ for any $e \in Env$ and $e_3 = e[pc \mapsto P]$ where $P = e(pc) \cap SP(n_0) \cap (\bigcap_{y \in V(E)} e(y))$, and thus $(e, e_3) \in \mathrm{MOVP}(n_0, n_3 n_2)$ when $e = proj(s, D)$ and $e_3 = proj(s_0, D)$. Moreover, $n_4 \Rightarrow \epsilon$ and $(e_4, e_5) \in \mathrm{MOVP}(n_4, \epsilon)$ for any $e_4 \in Env$ and $e_5 = e_4[x \mapsto e_4(x) \cap e_4(pc) \mid x \in W]$, and thus $(e_4, e_5) \in \mathrm{MOVP}(n_4, \epsilon)$ when $e_4 = proj(s_1, D)$ and $e_5 = proj(s_2, D)$. $\mathrm{MOVP}(n_0, n_1) \supseteq g_1(id, \mathrm{MOVP}(n_0, n_2)) \otimes \mathrm{MOVP}(n_2, n_1)$ and $(e, e'') \in g_1(id, \mathrm{MOVP}(n_0, n_2))$ where $e'' = proj(s'', D)$ since g_1 forces $e''(pc) = e(pc)$. It concludes that $n_0 \Rightarrow^* n_1$ and $(e, e') \in \mathrm{MOVP}(n_0, n_1)$.

(D) Consider the case that $C =$ **test** R **then** S_1 **else** S_2. Because of the symmetricalness of the definition, we assume that $R \subseteq D$ without loss of generality. By the definition of $\Downarrow$, $(S_1, s) \Downarrow_D^{SP(S)} s''$. By the inductive hypothesis, $n_3 \Rightarrow^* n_4$ and $(e, e'') \in \mathrm{MOVP}(n_3, n_4)$ where $n_3 = head(S_1)$, $n_4 = last(S_1)$, $e = proj(s, D)$, and $e'' = proj(s'', D)$. By the definition of W_π, $n_0 \Rightarrow n_3\, n_2$ and $(e, e) \in \mathrm{MOVP}(n_0, n_3 n_2)$ for $e = proj(s, D)$ since $R \subseteq D = e(dp)$. Moreover, $n_4 \Rightarrow \epsilon$ and $(e'', e'') \in \mathrm{MOVP}(n_4, \epsilon) = id$, and thus $n_0 \Rightarrow^* n_2$ and $(e, e'') \in \mathrm{MOVP}(n_0, n_2)$.

(E) If $C =$ **test** R **for** x, then by the definition of $\Downarrow$, $R \subseteq proj(s(x))$ and $s = s''$. By the definition of W_π, $n_0 \Rightarrow n_2$ and $(e, e) \in \mathrm{MOVP}(n_0, n_2)$ for $e = proj(s, D) = proj(s'', D)$ since $R \subseteq e(x) = proj(s(x))$.

□

Pablo Lamilla Alvarez received the M.S. degree in computer science and engineering from the Polytechnic University of Valencia in 2010. In 2011 he joined the Special Scholarship Program in Kochi University of Technology for PhD students. At the same time, he also joined the Junior Research Associate (JRA) program in the RIKEN institute from April 2011 until March 2012. His research interests are focused in the areas of formal methods and verificacion of software systems.

Yoshiaki Takata received the Ph.D. degree in information and computer science from Osaka University in 1997. He was with Nara Institute of Science and Technology as an Assistant Professor in 1997–2007. In 2007, he joined the faculty of Kochi University of Technology. His current research interest include formal specification and verification of software systems.