Doctorate thesis

Verification Model and Approximation-Based Implementation of Information-Based Access Control

1156010 Pablo LAMILLA ALVAREZ

Advisor Yoshiaki TAKATA

March 12, 2014

Course of Information Systems Engineering

Graduate School of Engineering, Kochi University of Technology

Abstract

Verification Model and Approximation-Based Implementation of Information-Based Access Control

Pablo LAMILLA ALVAREZ

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 and also the semantics of the IBAC model are relatively recent and complicated to be easily implemented in a real environment. Related to the first matter, this thesis 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. Related to the second matter, in this thesis we also propose an algorithm to approximate a subset of IBAC programs to a program that uses HBAC semantics. Accomplishing this, an IBAC program from the subset we defined could be implemented in an environment that is supported by HBAC.

key words Access Control Models, Model-Checking, Pushdown Systems

Contents

List of Figures

Chapter 1

Introduction

1.1 Research background

Security has been always an important matter in computer environments. Different ways and methods to protect important resources from non-authorized subjects have been developed and investigated during the last decades. For example, all the operating systems permits the user to statically assign different levels of permissions to any file in order to allow just a group of authorized users to write, read or execute the file. However, allowing or not the operation to be executed is decided always statically, i.e. for a given program that tries to access a file, the decision of allowing or not the program to access that file does not depend on the execution path of that program. The OS always allows or always forbids the operation to be executed, if the permissions are not modified. But, what happens if we want a program to access or not a file depending on the execution path of that program? For example, a given program that statically has access to a given file but the program calls some external libraries that may be dangerous for the file. In that case we should not allow the program to access the file, because the external libraries may damage the file. However, if the program does not access any dangerous external libraries, we should allow the program to access the file. This way of protecting a file is performed by what is called Access Control Model.

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 [3] 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 enough permissions to execute that operation. Following the example of the last paragraph, a program calls an external library that we do not trust and that library tries to execute a protected operation. Then the SBAC checks that the block code of the library does not have enough permissions to execute that operation and aborts it. However, what happens if, after the dangerous library is called, the main program tries to execute the operation? In this case the SBAC allows the program to execute the operation, which would be correct if the external library does not have any influence on that operation. However, the library could have decided the name of the file that is going to be written. In that case the SBAC allows a non-trusted subject to influence in a protected operation. 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. If we use the HBAC on our previous example, the protected operation will be always aborted if the external library is called before the operation. Note that HBAC protects the operation no matter if the library has influence or not on that operation. Because of this, 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 keeps track dynamically of the permissions of each single variable of the program, not just the permissions of blocks of code. Therefore, using the permissions of each variable, the IBAC verifies that only the code responsible for a security-sensitive operation is sufficiently authorized.

1.2 Objectives of the thesis

This thesis focuses on the IBAC model and treats two different issues regarding the IBAC model.

The first one refers to a formal verification, also called model checking, of the IBAC model. Model checking is explained in Chapter 2, but basically is a method to automatically and exhaustively verify that a program is free of bugs and meets the requirements we defined. In order to perform model checking of a program, we need a mathematical model to represent that program, in our case a program that uses the IBAC model. In this thesis we propose a formal model for IBAC programs suitable for formal verification. We model the behavior of an IBAC program as extended weighted-pushdown systems (EWPDS) [7]. This is an extension of the mathematical model called pushdown systems (PDS). Proposing a model for model-checking of IBAC programs is the first contribution of this thesis.

The second contribution of this thesis is related to the implementation of the IBAC in a real programming environment. At now, only the SBAC model is implemented on real environments, even though the HBAC and especially the IBAC models are theoretically safer. In this thesis we propose a method to approximate a subset of IBAC programs to the HBAC model. If a program can be transformed to an HBAC program, the original IBAC program can be implemented in a real system that supports the HBAC. An approximation to the SBAC model, which is the one that is implemented in a real system, would be ideal but the SBAC model semantics are much different from the IBAC ones and difficult to simulate the IBAC model. For this reason, we consider an approximation of IBAC to the HBAC model. For performing this approximation, we propose and define an algorithm that has an IBAC program as an input and produces a HBAC program with the same behavior as the input IBAC program.

This thesis is organized as follows: Chapter 1 presents a short introduction of the thesis as well as describes the objectives and the related research. Chapter 2 explains the three access control models mentioned before and presents a summary of what is model checking and the pushdown systems. Chapter 3 describes in detail the method and the modeling used for the formal verification problem of IBAC. In Chapter 4 we formally present the algorithm desgined to approximate an IBAC program to an HBAC program as well as some examples. Finally Chapter 5 concludes the thesis and proposes the future work.

1.3 Related Work

The first objective of this thesis extends across two different fields, i.e. formal verification and computer security. Specifically, researches that make use of the mathematical model pushdown systems in order to resolve security related problems are very similar to ours. In [18], WPDS model-checking is used to develop a distributed certificate-chain-discovery algorithm for a trust management system called SPKI/SDSI. In [17] 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 [17] as a key technique for precise non-interference analysis.

Moreover, the verification problem for other access control models, HBAC in these cases, has been discussed in works such as [5] and [4]. In these works they also solve the

formal verification problem by using the PDS structure. However, no formal verification has been proposed for IBAC, and therefore our work is the first to discuss and implement the model-checking problem for IBAC programs.

Regarding the second objective of this thesis, there are no works that achieve the implementation of the IBAC model. However, a few researches implement the HBAC model in real environments. In [15] Edjlali et al. describe the design and implementation of an HBAC mechanism for Java called *Deeds*. In [16] Krukow et al. present an implementation of a framework that provides both HBAC and reputation systems for Java programs. Finally in [14] Martinelli et al. propose solutions for improving the Java native security support by applying the HBAC on the Java environment. Our approximation method could be integrated with these HBAC implementations.

Chapter 2

Preliminaries

In this chapter the preliminary knowledge necessary for understanding this thesis is explained. As mentioned in the last chapter, this thesis extends across two different fields: computer security, specifically access control models in programming environments, and formal verification or model checking. This chapter explains the details of the three access control models mentioned in the introduction and also presents an overview of the specific part of the model checking field used in this thesis.

2.1 Access Control Models

2.1.1 Stack-Based Access Control

The Stack-Based Access Control model or SBAC is chronologically the first access control model among the three models considered in this thesis. It is implemented in programming environments such as Java or .NET. The objective of this and all the access control models is to protect security-sensitive operations in the program, like reading or writing a file, from dangerous or non-authorized functions. In order to achieve this, the SBAC allows the user to assign a set of permissions to each procedure. From now on, we assume that the permissions elements are atomic and without any hierarchy. This set is called *static permissions* and represents the grade of authorization of each procedure, i.e. which operations that procedure can perform or not. The static permissions do not change during the execution of the program. On the other hand the runtime

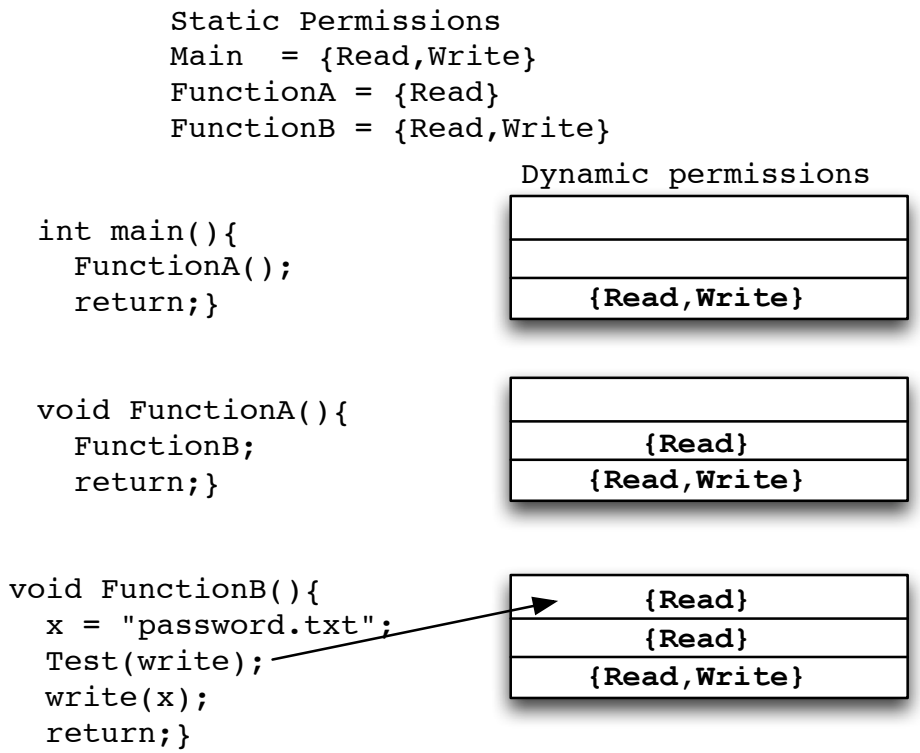

Fig. 2.1 Program example using SBAC.

system dynamically mantains the current permissions, called dynamic permissions, of the execution process. This set of permissions is dynamically updated during the execution of the program based on the static permissions of each called procedure. The dynamic permissions try to keep track of the execution path of the program and the communication between different procedures. This set of permissions is maintained in a stack-based style. When a procedure is called, the current dynamic permissions of the process are intersected with the static permissions of the called procedure. The result of this operation is pushed on the stack and becomes the current dynamic permissions. When the procedure finishes the current dynamic permissions are popped from the stack and the current dynamic permissions become the previous element on the stack.

Figure 2.1 shows an example how the SBAC model prevents a non-authorized access. In this example, both main function and FunctionB are trusted procedures whereas

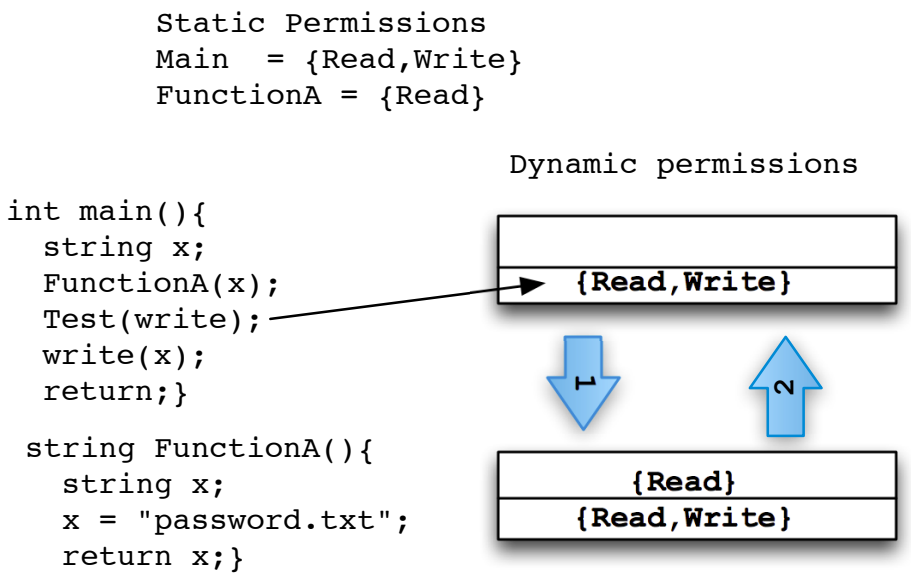

Fig. 2.2 Program example of a security fail in SBAC.

FunctionA is a non-trusted procedure. The user assigns the static permissions to each procedure depending if they are trusted or not. In this case, because the user does not trust FunctionA, just reading permissions are assigned to it. Then, during the execution, the dynamic permissions are maintained as the stack representation shows. First, the current permissions are the same as the static permissions of the main function. Then, when FunctionA is called, the current set of dynamic permissions {Read,Write} is intersected with the static permissions of FunctionA {Read}, producing the new current permissions {Read} and placing them into the stack. When FunctionB is called the same process is repeated producing the new current set of permissions {Read} and placing it into the top of the stack. Finally, when FunctionB wants to perform the write operation, the SBAC performs a test operation, previously placed by the user, in order to protect the write operation. This test checks if permission for writing is included in the current dynamic permissions which are placed at the top of the stack. In this example, the test operation fails because FunctionB has been called by FunctionA which does not have writting permissions. Therefore, the SBAC aborts the execution and the

write operation is not performed.

However, as mentioned before, SBAC does not keep track of information about previously executed methods. When a procedure finishes the current dynamic permissions are popped from the stack. As a result, the information of the level of authorization of previously executed methods is lost. This could cause a security fail in a SBAC program because a previously executed non-authorized function could influence a security-sensitive operation performed by a authorized function. This problem is shown in the example of Figure 2.2. In this example, a non-trusted procedure FunctionA decides the file that is going to be written by the write operation in the main function. Therefore, from a security view the operation should not be allowed. However, at the moment of the test operation, FunctionA finished its execution so the stack structure already lost the information of the level of authorization of FunctionA. Therefore, the test operation succeeds because the current dynamic permissions at that point are {Read,Write}, and then the write operation is performed, causing a security fail.

A solution for this problem is proposed by the authors of the next access control model explained in the following subsection.

2.1.2 History-Based Access Control

The History-Based Access Control model (HBAC) was informally introduced by Abadi et al. in [1] in order to solve the problem in the SBAC model explained in the previous subsection. The HBAC model works in a very similar way than the SBAC model but basically, the main difference lies in the treatment of the dynamic permissions. In contrast to the SBAC model, the HBAC does not keep the dynamic permissions in a stack, so in consequence the HBAC does not restore the current dynamic permissions to its previous state when a procedure finishes. The main implication of this change is that the HBAC keeps the history of previously executed methods and therefore, all

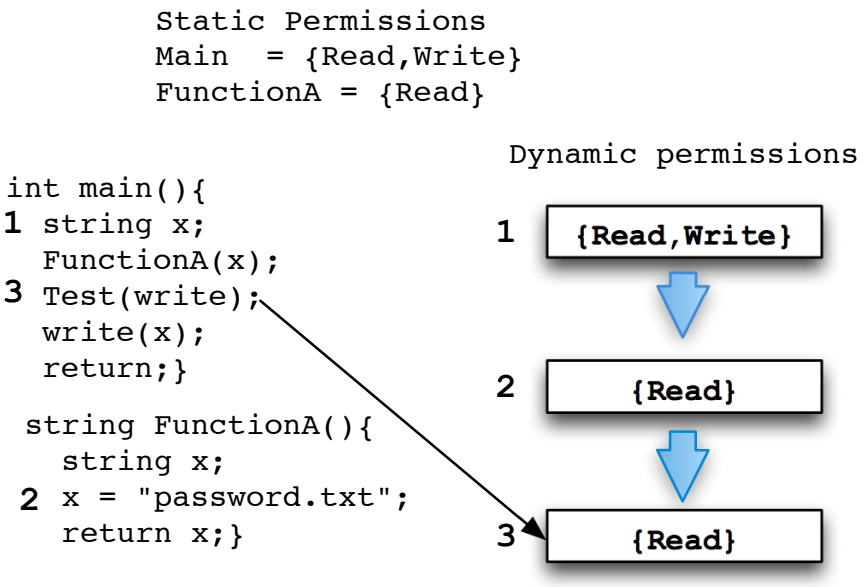

Fig. 2.3 Example of HBAC program that solves the SBAC security problem.

the code executed before a security-sensitive operation must have enough permissions to execute that operation. This behavior is shown in Figure 2.3. This example is the same as in Figure 2.2 but using the HBAC model. As we see in this figure, the current dynamic permissions are maintained after the FunctionA finishes. Then, when the test operation is performed (program point 3), because the current dynamic permissions do not include permission for writing, the test fails and the write operation is not allowed. Therefore, this is an example of how the HBAC model improves the security level of the SBAC model by keeping the history of previously executed procedures.

The HBAC also introduces two new operations related to the behavior of the dynamic permissions: *grant* and *accept*.

Grant(P,B) calls the function B with adding the set of permissions P, which must be a subset of the static permissions of the current function, to the dynamic permissions. This means that during the execution of the function B, the dynamic permissions are upgraded so that the set P is included in them. When the function B finishes, the

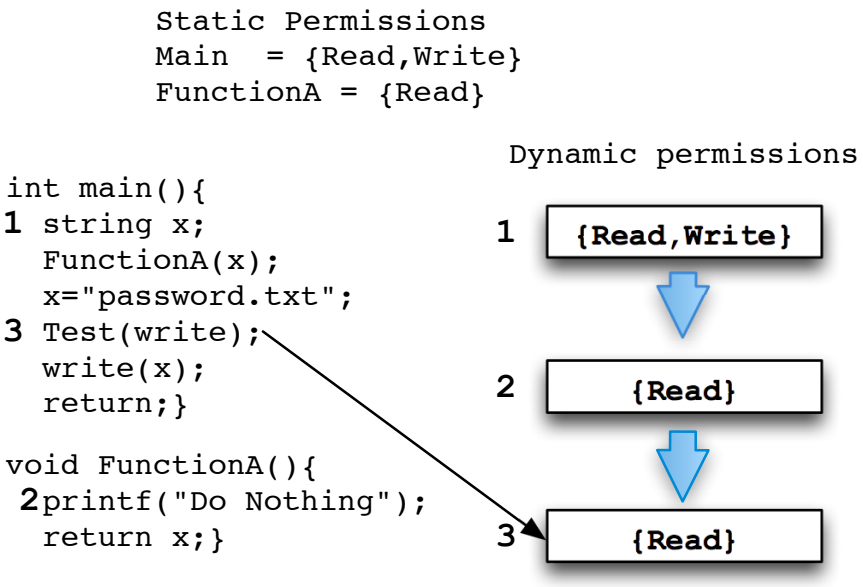

Fig. 2.4 Example of the high restrictiveness of the HBAC.

granted permissions are erased. Therefore, grant does not leave any extra permissions after the function B and preserves the loss of any permission during the function B.

Accept(P,B) preserves the set of permissions P in the dynamic permissions after the execution of function B. This means that, no matter which permissions are lost during the execution of B, the permissions in the set P will be included in the dynamic permissions after the execution of B. This command can be used to basically recover the SBAC behavior regarding the dynamic permissions.

As we have seen in this subsection, the HBAC protects security-sensitive operations in the cases SBAC fails. However, HBAC in some cases aborts operations that should have enough authorization level to be executed. The example in Figure 2.4 illustrates this problem. This program is similar to the example a of Figure 2.3 but in this case the FunctionA does not have any influence on the variable x and the write operation. However, because this function is called before the write operation, HBAC keeps the history of that call. Therefore, the dynamic permissions reflect that history the moment

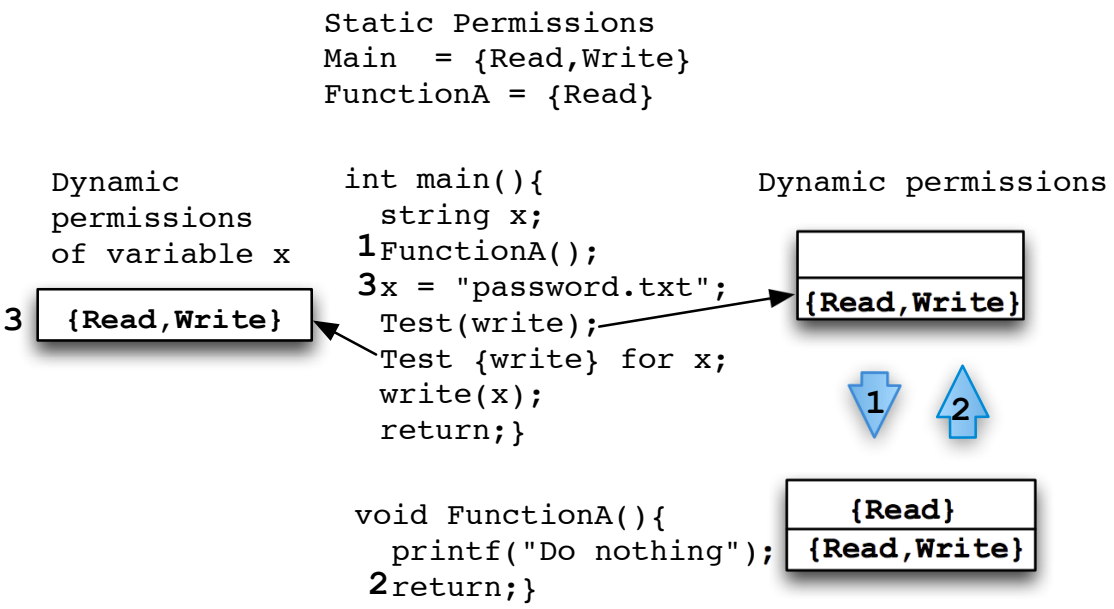

Fig. 2.5 Example of how the IBAC model solves the excessive restrictiveness of HBAC.

the test is executed (program point 3), preventing the write operation to be performed, even though FunctionA does not manipulate the variable x. The last model explained in this chapter, the IBAC, addresses this problem as we will see in the next subsection.

2.1.3 Information-Based Access Control

The Information-Based Access Control model (IBAC) was formally presented in 2007 by Pistoia et al. in [2] in order to solve the excessively restrictiveness of HBAC while keeping the same level of safety. The model follows the same characteristics of the previous models and also copies the SBAC treatment of the dynamic permissions. However it introduces a new element: a set of dynamic permissions for each variable in the program. This set changes every time the variable is updated. When this happens, the permissions of the variable are updated to the static permissions of the function that updates the variable. Therefore, IBAC model tracks during the whole execution the information about which function had influence in which variable. In consonance

with this change, the IBAC also introduces a new test statement to check if a set of permissions is included inside the dynamic permissions of a given variable. This test statement would be placed just before a security-sensitive operation that uses the given variable. We can see an example of the new elements introduced in the IBAC in Figure 2.5. In this figure the IBAC model is used in the program in Figure 2.4. The IBAC model recovers the stack structure to treat the current dynamic permissions and introduces a new dynamic permissions for the variable x, which are updated at every assignment of this variable. As we see in Figure 2.5 the behavior of the current dynamic permissions is exactly the same as in the SBAC model, then we can say that the IBAC model does not retain information about previously executed methods. However, what the IBAC models maintains is the information about which variables were updated by previously executed methods. This is kept in a dynamic set of permissions for each variable, in this example, just for variable x, which is updated at the assignment in the main function (program point 3). Then, before the write operation, two tests are performed. First the IBAC model tests if the permission for writing is included in the current dynamic permissions, which succeeds. Then, the second test checks if the permission for writing is included in the set of dynamic permissions of variable x, because is the variable that is going to be written. This second test also succeeds and the write operation is allowed to be performed. With this small example, we see how the IBAC model reduces the restrictiveness of the HBAC model while keeping the same level of security.

In this subsection we informally presented an overview of the three access control models related to this thesis. A deeper and more formal description of the IBAC model, including its original syntax and semantics, is presented in Chapter 3. Next subsection provides a shallow summary of the concept of model checking and the mathematical model used in Chapter 3 for formal verification of IBAC.

2.2 Model-Checking

Model checking is a verification technique that automatically and exhaustively explores all the states in a given system. This technique is normally used to verify some requirements or proterties that a system must meet. Generally, in order to use model checking we need first to model the system we want to verify using a mathematical model, for example finite state machines. Then we have to define some requirements that the system should meet and formalize them to produce a formal property specification. Examples of requirements could be that the system does not reach a deadlock, or the system is always running, etc. Finally, a model checking algorithm would take the system and the requirements as inputs and produce an output that will tell us if the property is satisfied for all the states in the systems or, on the contrary, exists some path in the system that violates the property.

Focusing in our specific case, the mathematical model chosen for representing an IBAC program is called Extended Weighted Pushdown Systems (EWPDS) which is an extension of the Pushdown System (PDS). The formal definitions and characteristics of this model are presented in Chapter 3, but basically a PDS is a stack-based transition system whose stack's length is not bounded. They are more expressive than finite state systems and because we can use the stack to keep track of active procedure calls, they become a natural model for recursion and procedure calls.

Finally, the property we aim to verify in an IBAC program is a security related requirement. In the formal verification field, a security property means that an specific dangerous or invalid state in a given system is never reached. Applying this to a given IBAC program, we could check if a given state that causes a security fail in the program is reached, due to a bug or a bad design, or on the contrary, the state is never reached and thus the program is safe.

The description of the method used in this thesis to combine the mathematical model EWPDS and the IBAC model in order to use model checking on IBAC programs is presented in the next chapter.

Chapter 3

EWPDS-based IBAC model

In this chapter we describe the process followed to model the IBAC by using the extended-weighted pushdown systems. This chapter is structured as follows: first we present the objective and motivations for this research. Second, the original syntax and semantics of the IBAC are described. Third, the formal definitions of the EWPDS are presented. Then, we show the design of our model, some examples of its usage and also we discuss the model-checking problem of our EWPDS model. Finally, we describe an implementation of our model using existing tools for EWPDS model-checking.

3.1 Objective and motivation

As mentioned in previous chapters, the main objective of this research is to use model-checking on programs modelled with the IBAC in order to find bugs or errors in the design that may cause a security fail. In order to achieve this, we design a EWPDS of the IBAC model, which is a suitable model for model-checking. The IBAC model is chosen for this research because is the most complete and the newest access control model, which means that exist some security policies that can be easily modeled using the IBAC model and cannot be modeled using earliers access control models. An example of this is the Resurrecting Duckling policy explained as follows.

Resurrecting Duckling policy[9] 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.

```
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. 3.1 Program Example that Models the Resurrecting Duckling Policy.

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 3.1 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 *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 Figure 3.1, 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.

3.2 Original syntax and semantics of IBAC

We review the syntax and the semantics of an IBAC program defined in [2]. Figure 3.2 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 of permissions, a procedure, and a variable, respectively.

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

Fig. 3.2 IBAC language syntax of commands.

For the convenience of the definition, we modify the syntax of a command sequence S in Figure 3.2 as $S ::= n \mid n{:}\, C; S$ 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 $head(n{:}\, C; S) = n$. Similarly, the last program point of S is denoted as $last(S)$; i.e., $last(n) = n$ and $last(n{:}\, C; S) = last(S)$. 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 Figure 3.3. 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*.

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,

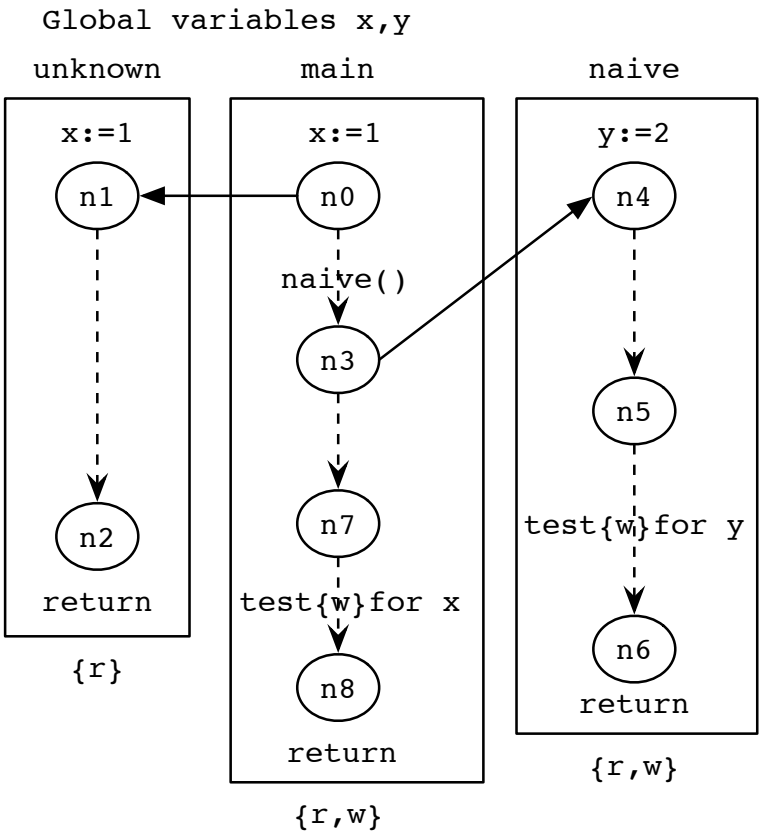

Fig. 3.3 A basic IBAC program.

$(E, s) \Downarrow_D^P R[v]$ 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:

$$\frac{p() = R[S], \quad (S, s)\Downarrow^{R}_{D\cap R} s'}{(p(), s)\Downarrow^{P}_{D} s'} \tag{3.1}$$

$$\frac{(S_1, s)\Downarrow^{P}_{D} s_1, \quad (S_2, s_1)\Downarrow^{P}_{D} s'}{(S_1; S_2, s)\Downarrow^{P}_{D} s'} \tag{3.2}$$

$$\frac{R \subseteq D, \quad (S_1, s)\Downarrow^{P}_{D} s'}{(test\ R\ then\ S_1\ else\ S_2, s)\Downarrow^{P}_{D} s'} \tag{3.3}$$

$$\frac{R \not\subseteq D, \quad (S_2, s)\Downarrow^{P}_{D} s'}{(test\ R\ then\ S_1\ else\ S_2, s)\Downarrow^{P}_{D} s'} \tag{3.4}$$

$$\frac{(E, s)\Downarrow^{P}_{D} P'[v], \quad R \subseteq P'}{(test\ R\ for\ E, s)\Downarrow^{P}_{D} s} \tag{3.5}$$

$$\frac{(S, s)\Downarrow^{P}_{D\cup(R\cap P)} s'}{(grant\ R\ in\ S, s)\Downarrow^{P}_{D} s'} \tag{3.6}$$

$$\frac{(E, s)\Downarrow^{P}_{D} R[v]}{(x := E, s)\Downarrow^{P}_{D} s[x \mapsto s(pc) \cap P \cap R[v]]} \tag{3.7}$$

$$\frac{\begin{array}{l}(E, s)\Downarrow^{P}_{D} R[false], \quad s_0 = s[pc \mapsto s(pc) \cap R] \\ (S_2, s_0)\Downarrow^{P}_{D} s_2, \quad V = write\_oracle(S_1, s) \\ s' = taint(s_0(pc), V, s_2)\end{array}}{(if\ E\ then\ S_1\ else\ S_2, s)\Downarrow^{P}_{D} s'[pc \mapsto s(pc)]} \tag{3.8}$$

$$\frac{\begin{array}{l}(E, s)\Downarrow^{P}_{D} R[true], \quad s_0 = s[pc \mapsto s(pc) \cap R] \\ (S_1, s_0)\Downarrow^{P}_{D} s_1, \quad V = write\_oracle(S_2, s) \\ s' = taint(s_0(pc), V, s_1)\end{array}}{(if\ E\ then\ S_1\ else\ S_2, s)\Downarrow^{P}_{D} s'[pc \mapsto s(pc)]} \tag{3.9}$$

Rules 3.1 and 3.2 define the behavior for the procedure call command and for a command sequence respectively. Rules 3.3 and 3.4 are the rules for the dynamic permission test statement, when it succeeds and when it fails respectively. Rule 3.5 defines the test for the permissions of a variable. In this case there is no "else" branch.

Rule 3.6 defines the semantics for the grant operation and rule 3.7 is the rule for the assignment statement. Finally rules 3.8 and 3.9 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.

3.3 Extended Weighted Pushdown System definitions

For a given program π, we model the transition system that represents the behavior of π as a *extended weighted pushdown system* (EWPDS)[13].

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* $w : \Delta \rightarrow D$ *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 WPDS. 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 WPDS 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 WPDS 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 WPDS 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)[7], 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 $g : \Delta_2 \rightarrow G$ *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:

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

- For a procedure call:

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

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.

3.4 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.

$$\frac{n' = head(S)}{\Delta(n{:}\, C; S) = \Delta(n{:}\, C; n') \cup \Delta(S)} \tag{3.10}$$

$$\begin{aligned} &t = n \hookrightarrow n' \in \Delta(n{:}\, x := E; n') \\ &\quad w(t) = \{(e, e[x \mapsto P]) \mid e \in Env \\ &P = (\textstyle\bigcap_{y \in V(E)} e(y)) \cap SP(n) \cap e(pc)\} \end{aligned} \tag{3.11}$$

$$\frac{m = head(p)}{\begin{aligned} &t = n \hookrightarrow m\, n' \in \Delta(n{:}\, grant\ R\ in\ p(); n') \\ &\quad w(t) = \{(e, e[dp \mapsto D]) \mid e \in Env \\ &D = (e(dp) \cup R) \cap SP(p)\} \quad g(t) = g_2 \end{aligned}} \tag{3.12}$$

$$\frac{p \in PR, \quad m = last(p)}{t = m \hookrightarrow \epsilon \in \Delta(m) \qquad w(t) = id} \tag{3.13}$$

$$\begin{aligned} &t = n \hookrightarrow n' \in \Delta(n{:}\, test\ R\ for\ x; n') \\ &\quad w(t) = \{(e, e) \mid e \in Env,\ R \subseteq e(x)\} \end{aligned} \tag{3.14}$$

$$\frac{\begin{aligned} &i, j \in \{1, 2\}, \quad i \neq j, \quad m = head(S_i) \\ &m' = last(S_i), \quad W = write\_oracle(S_j) \end{aligned}}{\begin{aligned} &t_1 = n \hookrightarrow m\, n' \in \Delta(n{:}\, if\ E\ then\ S_1\ else\ S_2; n') \\ &\quad w(t_1) = \{(e, e[pc \mapsto P]) \mid e \in Env, \\ &\qquad P = e(pc) \cap SP(n) \cap (\textstyle\bigcap_{y \in V(E)} e(y))\} \\ &\qquad\quad g(t_1) = g_1 \\ &t_2 = m' \hookrightarrow \epsilon \in \Delta(m') \\ &\quad w(t_2) = \{(e, e[x \mapsto e(x) \cap e(pc) \mid x \in W]) \\ &\mid e \in Env\} \\ &\Delta(n{:}\, if\ E\ then\ S_1\ else\ S_2; n') \supseteq \Delta(S_1) \cup \Delta(S_2) \end{aligned}} \tag{3.15}$$

$$\frac{m = head(S_1), \quad m' = last(S_1)}{\begin{array}{l} t_1 = n \hookrightarrow m\, n' \in \Delta(n{:}\, test\ R\ then\ S_1\ else\ S_2; n') \\ \quad w(t_1) = \{(e,e) \mid e \in Env,\ R \subseteq e(dp)\} \\ \quad\quad g(t_1) = g_0 \\ t_2 = m' \hookrightarrow \epsilon \in \Delta(m') \qquad w(t_2) = id \\ \Delta(n{:}\, test\ R\ then\ S_1\ else\ S_2; n') \supseteq \Delta(S_1) \cup \Delta(S_2) \end{array}} \tag{3.16}$$

$$\frac{m = head(S_2), \quad m' = last(S_2)}{\begin{array}{l} t_1 = n \hookrightarrow m\, n' \in \Delta(n{:}\, test\ R\ then\ S_1\ else\ S_2; n') \\ \quad w(t_1) = \{(e,e) \mid e \in Env,\ R \not\subseteq e(dp)\} \\ \quad\quad g(t_1) = g_0 \\ t_2 = m' \hookrightarrow \epsilon \in \Delta(m') \qquad w(t_2) = id \\ \Delta(n{:}\, test\ R\ then\ S_1\ else\ S_2; n') \supseteq \Delta(S_1) \cup \Delta(S_2) \end{array}} \tag{3.17}$$

Rule (3.10) defines the set of transition rules for a command sequence.

Rule (3.11) 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 (3.12) 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 (3.13) 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 (3.12). 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 (3.15) 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 WPDS 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 (3.16) and (3.17) 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 (3.14) 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.

3.4.1 Model Example

Let us return to the IBAC program π_1 in Figure 3.3. When the unknown procedure is called by n_0, the current dynamic permissions i.e., the variable *dp* in the weight of the WPDS, 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

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. 3.4 Transitions of $W_{\pi 1}$.

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 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 WPDS $W_{\pi 1}$ and the weight after each transition following the semantics explained before are shown in Figure 3.4.

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.

3.5 Formal Verification problem

Let us discuss in this section the model-checking problem of our WPDS model. Let π be an IBAC program and W_π be the WPDS 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:

$$(e_0, e) \notin MOVP\,(n_0, n\xi) \;\; for\ any\ e\ and\ \xi.$$

This expression means that when the WPDS 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_1 in Figure 3.3. Following the original semantics of the IBAC model, in program π_1 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 $\mathrm{W}_{\pi 1}$, 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_1 is maintained in the WPDS $\mathrm{W}_{\pi 1}$.

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 [8], we can obtain a WPDS 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 WPDS 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 WPDS W_π satisfies $n_0 \Rightarrow^* n_1$ and $(e, e') \in MOVP(n_0, n_1)$ 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 Appendix 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 WPDS, if W_π includes conditional clauses, its reachable set of states is greater than the program π.

3.6 Implementation

The model presented in this thesis is implemented using the model-checker for WPDS called WALi[10]. This tool provides a C++ interface for easily creating and verifying WPDS and also provides an add-on that implements a binary relation domain using the Binary Decision Diagram (BDD) library Buddy[11]. The implementation of a binary relation includes the basic semiring methods that WALi needs in any weight domain. These methods are the followings:

- *One()*. Returns the neutral element 1 of the semi-ring. In our model, the rules whose weight is *id* will return this element.
- *Zero()*. Returns the empty element 0 of the semi-ring. This method will not be used explicitly in our model, though a weight at some configuration will become equal to this element.
- *Combine()*. Returns the semi-ring that result of the union of two semi-rings. This

operation is used in our model at the end of a conditional clause, where two different paths converge.

- *Extend()*. The extend operation returns the composition of two semi-rings. This operation is used along all the elements of one single path in order to calculate the final weight of that path.

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 WPDS 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 section 3.3. The code of all these methods is shown in the Appendix B.

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 $\left|VR'\right| \times |PRM|$ 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_{Pa}, pc_{Pb}, dp_{Pa}, dp_{Pb}) 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 $(x_{pre}, x_{post}, ..., z_{pre}, z_{post})$,

```
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. 3.5 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. 3.6 Program test of grant and dynamic permission check with 2 permissions.

instead of $(x_{pre}, ..., z_{pre}, x_{post}, ..., z_{post})$. 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 WPDS of four IBAC programs, each one representing a group of IBAC programs. These four programs try

```
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. 3.7 Program test of loops using recursion with 2 permissions.

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 Figure 3.1. 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 code of this example is shown in the Appendix B.

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 Figure 3.5 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 Figure 3.6 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 increases.

Finally the third artifitial program shown in Figure 3.7 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

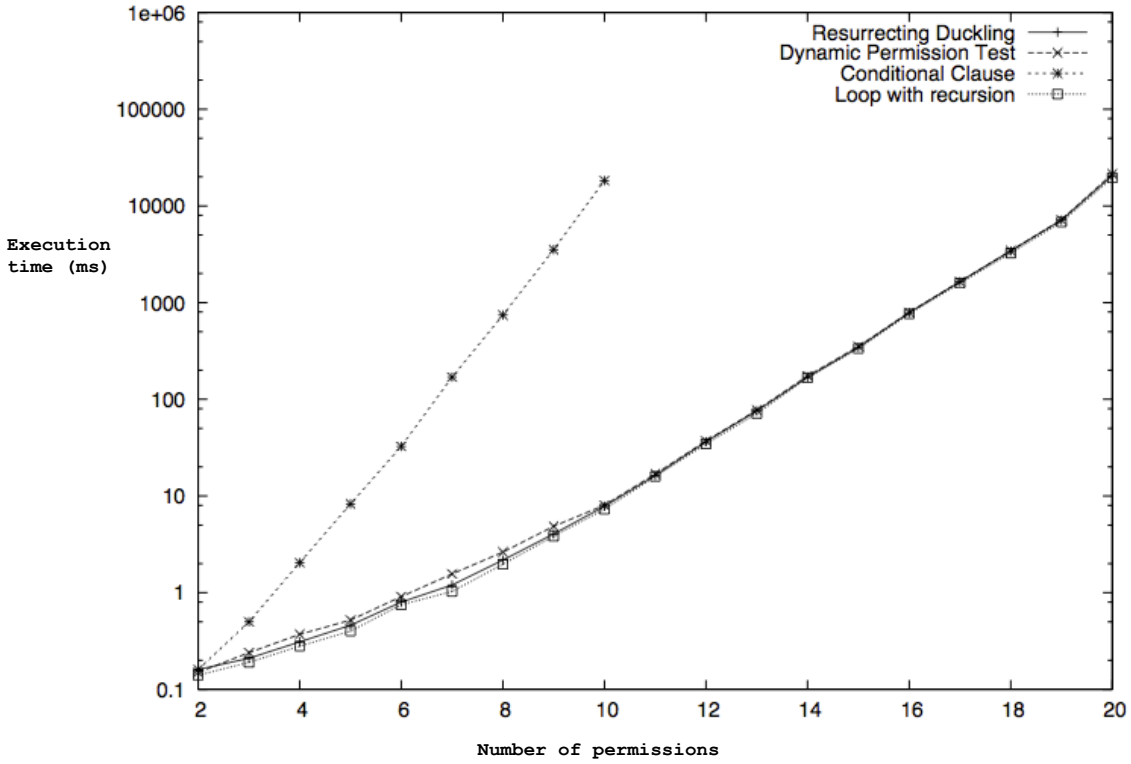

Fig. 3.8 Performance evaluation of a query in our WPDS model.

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 WPDS, a poststar query[13] 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 Figure 3.8. 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 Figure 3.1 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[12].

Chapter 4

HBAC-IBAC approximation

In this chapter we present the second research of this thesis which is an algorithm to approximate a program modeled by the IBAC to a program modeled by the HBAC. This chapter is structured as follows: first we present the objective and motivations for this research. Second, a detailed comparison between the HBAC and the IBAC models is described. Third, the approximation algorithm is formally presented and its steps described. Finally we explain a practical example using the algorithm and propose some improvements for decreasing the costing time of the algorithm.

4.1 Objective

The objective of this research is related to the IBAC implementation in a real programming environment. As mentioned in Chapter 1, just the SBAC model is implemented on real environments, even though the HBAC and especially the IBAC models are theoretically safer. In this chapter we propose an algorithm to transform as much as possible a subset of IBAC programs to the HBAC model. If exists an environment that supports the HBAC, an IBAC program that can be approximated to an HBAC program, could be used also in that HBAC-supported environment. Therefore, this way we could achieve an IBAC-supported environment. The HBAC model is chosen as the output of our algorithm because, even thought the SBAC is the only one that is implemented in a real system, the HBAC semantics are richer and closer to the IBAC model. Moreover

$variable = constant;$	assignment
$f();\ return;$	procedure call; return statement
$f()\ or\ g();$	nondeterministic choice procedure call

Fig. 4.1 IBAC subset syntax.

as shown in section 1.3, it exists a few works that try to create an implementation of the HBAC in a real environment, where our algorithm could be integrated in order to provide IBAC support to those environments.

Regarding to the input IBAC programs of the algorithm presented in this chapter, we limit the original syntax of the IBAC model to a narrower subset of programs. The syntax of our subset of IBAC programs is shown in Figure 4.1. We impose the following restrictions:

- No conditional clauses "if else". The indeterminism in the program is introduced by nondeterministic choice of procedure calls. This restriction is due to the difficulty of implementing with just HBAC semantics the complex operations that IBAC uses against implicit information flow[2].
- No loops statements such as "for" and "while". Iterations should be represented by recursive calls. This restriction is due to the absence of loop commands in the original IBAC syntax.
- The right-hand part of a variable assignment is always a constant value. This restriction simplifies the assignment operation problem in an IBAC program because the set of permission of the updated variable always gets the static permissions of the function where the assignment is executed. This way our algorithm is simplified also.

On the other hand, the objective of our approximation is to simulate an IBAC

program using the HBAC semantics while preserving the safety property from the original IBAC program. This means that any path that is aborted in the original IBAC program is aborted in its HBAC approximation program. This way, the HBAC approximation program does not introduce any new security violation that does not occur in the original IBAC program.

4.2 HBAC-IBAC comparison

Both HBAC and IBAC semantics and behaviors have been explained in previous chapters, but in this section we would like to highlight the main differences between the IBAC and the HBAC models regarding their semantics.

The first important difference is related to the amount of dynamic set of permissions controlled by both access control models. In the HBAC, only one set of dynamic permissions, the current permissions of the program, are kept during the execution of the code. However, the IBAC model introduces another set of dynamic permissions per variable in the program, as well as the current permissions of the program. Therefore, the IBAC model tracks more information during the execution of the program than the HBAC model. In order to keep the same behavior in our HBAC approximation than the IBAC model, our algorithm must increase in some way the amount of information the HBAC model is able to maintain.

The second difference between the HBAC and the IBAC is the behavior of the current dynamic permissions after a return statement. In the IBAC, when a function finishes the current dynamic permissions are restored to its previous value in the stack, like in the SBAC model. However, in the HBAC the set of permissions maintains its value after a return statement. Therefore, our algorithm has to produce the output HBAC modified in a way that it emulates the same behavior of the dynamic permissions

	HBAC	IBAC
Sets of dynamic permissions	1 set of current permissions	1 set of current permissions 1 set for each variable
Behavior of the current permissions at return statements	Keep the current permissions	Restore the current permissions to its previous value in the stack
Points of the program where the dynamic permissions are updated	Function calls	Function calls Return statements Variable assignments

Fig. 4.2 Main differences addressed by our algorithm between the HBAC and the IBAC.

after a return statement than in the IBAC model.

The last significance difference is the moment during the execution of the program when the dynamic permissions are updated. In the HBAC model, an update of the dynamic permissions just occurs when a procedure is called. However, in the IBAC model the dynamic permissions of each variable are also updated when the execution reaches an assignment of a variable. Because the HBAC model does not perform any action on the dynamic permissions at an assignment statement, the output program of our algorithm should force an update in the dynamic permissions when the execution reaches an assignment. In the next section is explained the method followed to achieve this objective.

A summary of the differences between the HBAC and the IBAC models is shown in Figure 4.2. The design process of the algorithm explained in the next section is focused

	IBAC	HBAC approx
1st Step Explode permissions	{A,B} Variable x,y	{A,B,xA,xB,yA,yB}
2nd Step Replace function calls	func(); SP of func = {A}	accept({A},func())

3rd Step Downgrade type assignments	Prm of x = {A,B}
	SP of func = {B}
	func(){
	x = 1;}
		SP of fx={B,xB}
		SP of func={B,xA,xB}
		DP = {B,xA,xB}
		fx(){}
		func(){
		x = 1;
		fx();}

4rd Step Upgrade type assignments	Prm of x = {}
	SP of func = {A}
	func(){
	x = 1;}
		SP of fx={xA}
		SP of func={A,xA}
		DP = {A}
		fx(){}
		func(){
		x = 1;
		grant({xA},fx();}
		PLUS **mirror algorithm** explained in section 4.3.1

Fig. 4.3 Steps of the HBAC approximation algorithm

in solving these differences between the models. Each step of the algorithm will address one of these differences so that the output program has the same behavior than the original program modeled by the IBAC but using the HBAC semantics.

4.3 Approximation Algorithm

This algorithm consists basically of four steps which are shown in the scheme of Figure 4.3. As mentioned before, each of these steps address one of the main differences between IBAC and HBAC explained in the previous section. After each step, the difference addressed is eliminated so that the output program modeled by the HBAC has the same behavior than the original program modeled by the IBAC. In our case, two programs have the same behavior if they allow and prevent the same security sensitive operations. This also means that the result of each check permission statements in both programs is also the same. As follows we informally explain each step of the algorithm.

The first step is to explode the permissions in the HBAC program in order to track the permissions of each variable. Specifically, we create new permissions xP for each variable x and permission P in the original IBAC program. For example, if the original IBAC program has two variables x and y and two permissions A and B, the permissions in our HBAC approximation would be exploded to A, B, xA, xB, yA, yB. The algorithm is designed so that the dynamic permissions in the resultant HBAC program contain xP for a variable x and a permission P if and only if in the input IBAC program, the dynamic permissions of x contain P at the corresponding program point.

In the second step we replace all the function call $func()$ in the original IBAC program with the HBAC operation $accept(SP, func())$, where SP is the set of static permissions of the function in the IBAC program where the function call statement exists. This step is done in order to simulate the behavior of the dynamic permissions in IBAC which is the same as in SBAC.

The third and the fourth steps are related to the same problem which is to create methods in the HBAC program for simulating the update of the permissions of each variable in the IBAC program. In IBAC the permissions of variables are updated at

assignment statements. However in HBAC, the dynamic permissions are only updated at function call statements (or return statements). Therefore we insert a function call statement in HBAC at each assignment statement so that the dynamic permissions of HBAC simulate the permissions of the updated variable of IBAC. According to the HBAC semantics, removing some permissions from the dynamic permissions (we call this case a "downgrade") is easy, while adding permissions to the dynamic permissions (we call this case an "upgrade") is much harder. Note that the behavior of an IBAC program can be modeled by a pushdown system (PDS)[8], and by analyzing the PDS, at each assignment statement we can know whether or not an upgrade possibly occurs.

For a downgrade (3rd step), we create a new function whose static permissions do not include the permissions to be removed, and we insert a call to that function to the HBAC program just after the assignment statement. This function call simply removes the specified permissions from the set of dynamic permissions. In Figure 4.3 for example, we want to remove the permission xA from the dynamic permissions, because in the original IBAC program the permission A is removed from the set of permissions of x. Therefore, we create and call a new function that does not include xA in its set of permissions. For an upgrade (4th step), only using the grant command we can add permissions to the dynamic permissions. Therefore, simillary as the previous step, we create and call a new function that using the grant operator to add the new permission we want. In Figure 4.3 this permission would be xA. However, the added permissions are erased when the function called by the grant command finishes. To solve this problem, we modify the structure of the program using the algorithm explained in the next subsection.

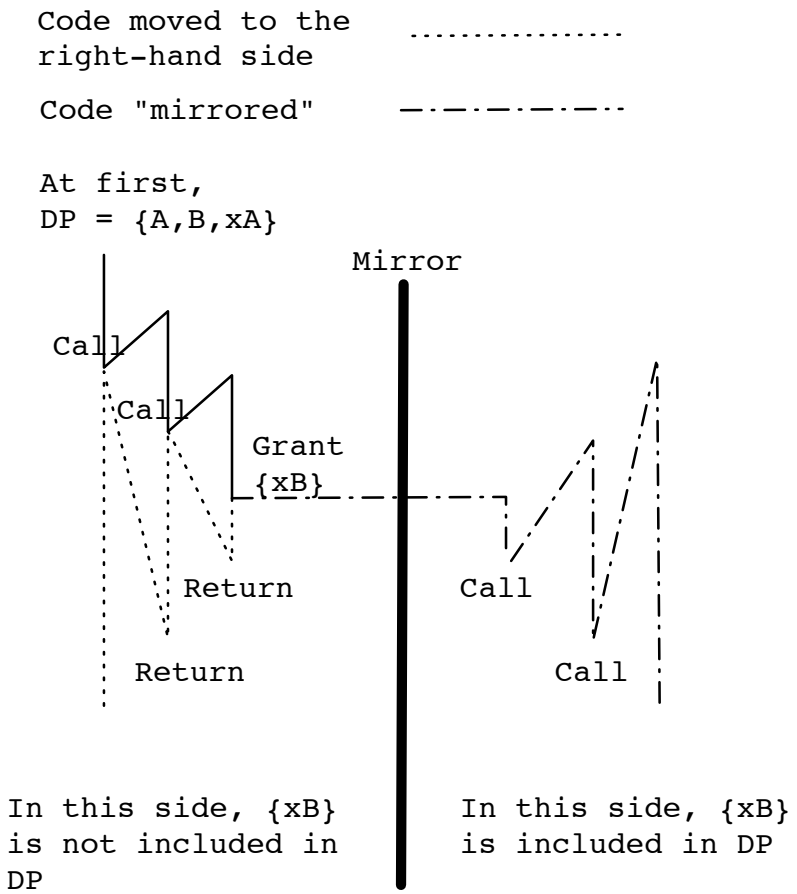

Fig. 4.4 Mirror algorithm diagram

4.3.1 Mirror algorithm

The purpose of this algorithm is to create a new program where the dynamic permissions are upgraded from a point in the program until the end of it. Because the permissions added with a grant operation are lost after the granted function finishes, we need to execute the rest of the code without returning from the granted function. The main idea of this algorithm is to move the rest of the code from the point where the grant operation is executed to the granted function. However, because we cannot return to the callee, we need to move the code in a "mirror" way, changing the return statements to call statements that call a "mirrored" function. Figure 4.4 shows a diagram of this method. Assume we want to add the permission B to a given variable x, which in

our approximation means adding the permission xB to the dynamic permissions. After the artificial grant(xB) we placed is called, it passes a barrier that cannot cross back, because as explained before, the permission xB would be eliminated from the dynamic permissions; we call this barrier the *mirror*. Then, the rest of the code from that point is placed at the right-hand side of the mirror, where the dynamic permissions are granted. Because we cannot cross back the mirror, the return statements of the callee functions on the left-hand side are replaced with call statements that call a mirror version of the original function. The code of these mirrored versions consists of the code that has not been executed before the artificial grant we placed. This way, we execute the same instructions of the moved code but in a context where the permission xB is included in the dynamic permissions, i.e. variable x has the permission B.

4.3.2 Formal representation

In the last section we informally explained the steps of our algorithm in order to be more understandable. In this section we try to formalize the steps of the algorithm using a pseudocode language. Figure 4.5 represents the approximation algorithm explained in the previous section.

The algorithm in Figure 4.5 receives a program modeled by the IBAC as an input and produces a program that, when modeled by the HBAC, has the same behavior as the input program. We call this output program HBAC approximation program. Then the algorithm proceeds we the steps explained in the previous section. First for each procedure of the program, the algorithm adds to the static permissions of each procedure the permissions elements for each variable. These new permissions represents if a variable has or not a given permission. Then the algorithm replaces all the call statements to procedures in the input program for the accept command in order to restore the dynamic permissions when a procedure finishes. Next, a small

input : IBAC program

output: HBAC approximation program

foreach *Function block* **do**
  Explode the static permissions to add the elements $x_n P_n$ where x is a variable and P is a permission in the original IBAC program;

end

foreach *Function call func()* **do**
  Replace the the call statement with accept(SP, func()), where SP are the static permissions of the current function where the call statement is;

end

foreach *Test P for x command* **do**
  Replace the check statement Test P for x, where P is a set of permissions and x a variable, for the statement Test xP where xP is a set of permissions that includes the permissions of P for the variable x;

end

foreach *variable x in the program* **do**
  while *not at end of the program* **do**
    Use the UD-chain algorithm for that variable to find the first UD pair (P1,P2);
    Let SP be the set of static permissions of the function where P1 is;
    Let DP be the dynamic permissions at P1;
    if *The permissions of the variable are downgraded* **then**
      Introduce a call function statement funcX_n() after P1;
      Create a new function funcx_i() with static permissions SP minus the elements of the variable permissions that are not included in the static permissions of the callee function;
    else
      Introduce a command grant(SPx, funcx_i) after P1 where SPx is the permissions we want to add to the variable x;
      Create a new function funcx_i() with static permissions SP;
      If not called befored, call the "mirror algorithm";
    end
  end

end

Fig. 4.5 HBAC-based approximation algorithm

step but necessary is to substitute the IBAC check statements $TestPforx$ with HBAC check statements $TestPx$, because the output program should be an HBAC modeled program. The elements of the set Px will be the same permissions of the set P but related to the variable x. For example, if we are checking the variable x and the set P includes the permissions A and B, the set Px will consist of the elements xA and xB.

The last steps of the algorithm are related to the treatement of the assignment statements in the original program. As mentioned in the last section, before applying the algorithm, we can use model checking on a PDS model of the IBAC program in order to know which assignments have the possibility to upgrade the permissions of the variable. Because the program may have more than one different execution path, and therefore the same assignment may be an upgrade in one path but a downgrade in another path, we always mark that assignment as an upgrade. This is done because the upgrade case is more general than the downgrade case and thus the upgrade case includes the downgrade case. We can mark every case as an upgrade but this will overuse the mirror algorithm which is the most expensive part of our algorithm, and therefore the upgrade cases should be as less as possible.

Regarding to this last matter, our algorithm also makes use of the UD-chain (use-definition) algorithm in order to search the last assignment before a test statement. In our case, the use of the variable would be a test statement of that variable and the definitions of the variable would be all the assignment statements of that variable before that test statement. This process is done because an assignment completely overwrites the variable permissions and thus deleting the influence of the last assignment. Therefore, only the last assignment before a test statement is relevant. Note that this only happens because we limited the original semantics of the IBAC in case of assignments so that the right-hand part of an assignment is always a constant variable.

Finally, the algorithm treats each assignment depending if they are marked as

```
input  : HBAC approximation program
output: HBAC approximation program mirrored
foreach Instruction from the initial point until the end of the program do
    if The instruction is not a return statement then
        Move the instruction line from the original function to the created function;
    else
        Insert a call statement "xMirrored()" in the created function;
        Create a new function called "xMirrored" where x is the name of the function that is
        going to be returned by the return statement;
    end
end
```

Fig. 4.6 Mirror algorithm

upgrade or not. If not the algorithm just creates and call a new function in order to eliminate the selected permission elements. If it is an upgrade, the algorithm creates and call using the grant command a new function in order to increase the permissions. Then we call the mirror algorithm in order to modify the structure of the program from the current point. The structure of the mirror algorithm is shown in Figure 4.6. The algorithm keeps two points in the program, one at the left side in the mirror which is the original function, and the other at the right side of the mirror which is the beginning of the created granted function. From these points, we move each instruction that are followed in the program from the left side to the right side of the mirror. If we encounter a return statement at the left side of the mirror, we have to create a new function at the right side of the mirror that will have the same instructions than the function returned by the encountered return statement. The function name will be the same name of the original function plus the word "Mirrored". Then, this function is called by the previous function in the right side of the mirror. These steps are followed until the end of the program.

The mirror algorithm completely modifies the structure of the program but we

```
Static Permissions
main  = {A,B}     LogoutA = {A,B}
LoginA = {A}      LogoutB = {A,B}
LoginB = {B}

       int main(){
         LoginA() or LoginB();
         LogoutA() or LogoutB();
         LoginA() or LoginB();
         return;}

LoginA(){             LoginB(){
  Test{A,B}for x;       Test{A,B}for x;
  x = 1;                x = 1;
  return;}              return;}

LogoutA(){            LogoutB(){
  Test{A}for x;         Test{B}for x;
  x = 1;                x = 1;
  return;}              return;}
```

Fig. 4.7 Input IBAC program

only need to use it if it is the first upgrade assignment we treat in the algorithm. After applying it the first time, since the mirror algorithm completely modifies the rest of the program from the point where the assignment is, from the next upgrade assignment we find that the rest of the program is already "mirrored". Therefore, we do not need to worry about losing the new permissions added by the grant command from the second upgrade assignment. We just need to move the rest of the code of the function where the upgrade assignment is to the new created and called function.

4.3.3 Example of the algorithm

In this section we will follow step by step a full example applying our algorithm to a IBAC program, shown in Figure 4.7. The program of this example represents a simple login system with two users A and B. First, user A or user B can log to the system by

```
Static Permissions
main  = {A,B,xA,xB}  LogoutA = {A,B,xA,xB}
LoginA = {A,xA,xB}   LogoutB = {A,B,xA,xB}
LoginB = {B,xA,xB}

        int main(){
          LoginA() or LoginB();
          LogoutA() or LogoutB();
          LoginA() or LoginB();
          return;}

LoginA(){            LoginB(){
  Test{A,B}for x;      Test{A,B}for x;
  x = 1;               x = 1;
  return;}             return;}

LogoutA(){           LogoutB(){
  Test{A}for x;        Test{B}for x;
  x = 1;               x = 1;
  return;}             return;}
```

Fig. 4.8 Intermediate output program after applying the first step of the algorithm

using the functions loginA or loginB respectively. When one of the logs in, the variable x loses the permission A or B respectively in order to represent which user is logged in. Then, a logout for user A or user B is called. This function checks that the user that is currently logged into the system can log out. Therefore, if loginA was called before, logoutB cannot be called and vice-versa. When a logout function executes correctly, variable x recovers both permissions A and B so that a login function can be called afterwards and thus a new user can log in to the system. This is an example of IBAC program where the permissions of a variable can be upgraded after being downgraded.

Let us start applying our algorithm to the program example. First of all, the static permissions of each procedure have to be exploded in order to include the variable permissions. Because in our example there is just one variable x, this first step is solved by adding the permissions xA and xB to all the set of static permissions. The result is

```
Static Permissions
main  = {A,B,xA,xB}  LogoutA = {A,B,xA,xB}
LoginA = {A,xA,xB}   LogoutB = {A,B,xA,xB}
LoginB = {B,xA,xB}

int main(){
  accept({A,B},LoginA())or accept({A,B},LoginB());
  accept({A,B},LogoutA())or accept({A,B},LogoutB());
  accept({A,B},LoginA())or accept({A,B},LoginB());
  return;}

  LoginA(){            LoginB(){
    Test{A,B}for x;      Test{A,B}for x;
    x = 1;               x = 1;
    return;}             return;}

  LogoutA(){           LogoutB(){
    Test{A}for x;        Test{B}for x;
    x = 1;               x = 1;
    return;}             return;}
```

Fig. 4.9 Intermediate output program after applying the second step of the algorithm

shown in Figure 4.8.

The second step of the algorithm is to substitute all the procedure calls with accept commands in order to restore the current permissions after a procedure finishes. With this change the HBAC model obtains the same behavior as the IBAC regarding the return statements. The result of this step is shown in Figure 4.9.

The third step of the algorithm is to treat the assignments that downgrade permissions of a variable, i.e, the variable loses permissions after the execution of the assignment statement. This happens in our program example at the login functions, where the variable x loses the permission xA or the permission xB. The algorithm forces the update on the dynamic permissions of the HBAC model by addin new functions x1 and x2 in order to eliminate the permissions xA and xB respectively. The result is shown in Figure 4.10.

```
Static Permissions
main  = {A,B,xA,xB}  LogoutA = {A,B,xA,xB} x1{A,xA}
LoginA = {A,xA,xB}   LogoutB = {A,B,xA,xB} x2{B,xB}
LoginB = {B,xA,xB}

int main(){
  accept({A,B},LoginA())or accept({A,B},LoginB());
  accept({A,B},LogoutA())or accept({A,B},LogoutB());
  accept({A,B},LoginA())or accept({A,B},LoginB());
  return;}

LoginA(){          LoginB(){           x1(){
  Test{xA,xB};       Test{xA,xB};        return;}
  x1();              x2();             x2(){
  return;}           return;}            return;}

LogoutA(){         LogoutB(){
  Test{A}for x;      Test{B}for x;
  x = 1;             x = 1;
  return;}           return;}
```

Fig. 4.10 Intermediate output program after applying the third step of the algorithm

The last step in our algorithm treats the assignments that upgrades permissions of a variable, i.e, the variable gains new permissions after the execution of the assignment statement. This case happens in both Logout functions after the test statement. In those cases the variable gets the permission xA or the permission xB. The algorithm first creates a grant command in order to add the permissions xA and xB. These grant command call a new function called LogoutAmirr in the user A case and LogoutBmirr in the user B case. Then, because the granted permissions would be lost if we return from these functions, we need to create and call a mirror version of the caller function of the logout functions, in this case the main function. Therefore, we create the mainMirr function which will continue the rest of the code of the original main function. This way, before we execute the login functions again, the permissions xA and xB are included in the dynamic permissions so that another user can log in correctly. The result of this

```
Static Permissions
main  = {A,B,xA,xB}  LogoutA = {A,B,xA,xB} x1{A,xA}
LoginA = {A,xA,xB}   LogoutB = {A,B,xA,xB} x2{B,xB}
LoginB = {B,xA,xB}   mainMirr = {A,B,xA,xB}
LogoutAmirr = {A,B,xA,xB} LogoutBmirr = {A,B,xA,xB}

int main(){
 accept({A,B},LoginA())or accept({A,B},LoginB());
 accept({A,B},LogoutA())or accept({A,B},LogoutB());
 return;}

LoginA(){       LoginB(){              x1(){
  Test{xA,xB};    Test{xA,xB};           return;}
  x1();           x2();                x2(){
  return;}        return;}               return;}


LogoutA(){       LogoutB(){           LogoutBmirr(){
  Test{xA};        Test{xB};            mainMirr();
  grant({xA,xB},   grant({xA,xB},       return;}
   LogoutAmirr()); LogoutBmirr());
  return;}         return;}           LogoutAmirr(){
                                        mainMirr();
                                        return;}
 int mainMirr(){
  accept({A,B},LoginA()) or accept({A,B},LoginB());
  return;}
```

Fig. 4.11 HBAC approximation program

step and the final output of the algorithm is shown in Figure 4.11. The program in this figure modeled by the HBAC model preserves the same behavior as the original program modeled by the IBAC because every test statement produces the same result in both programs, regardless the path taken during the execution.

4.3.4 Performance considerations

The algorithm shown in the previous sections achieves the objective we proposed for this research: it modifies an IBAC program in the way that, modeled by the HBAC model, it maintains the same behavior as the original IBAC program. However, the al-

gorithm dramatically modifies the original program, especially increasing the number of procedures due to the mirror algorithm. Because of this, the approximation HBAC program will be heavier than the original program if modeled by the HBAC. The execution time of this approximation HBAC program will depend mostly in the number of times we create a new function for the treatement of an assignment. However, the question is if this approximation program is heavier or lighter than the original program modeled by the IBAC. Because it does not exist a programing environment that supports the IBAC model, this cannot be proved but we can guess if we compare the performance results of the chapter 3 of this thesis with the results in the paper [5]. In this paper the authors design and implement a PDS-based model for the HBAC in order to perform model-checking. If we compare the results in that paper with the ones of our EWPDS-based model for the IBAC, we can see that the execution time of model-checking the IBAC model is much higher than the execution time of model-checking the HBAC model. Therefore, one can guess that directly implementing the IBAC model in a programming environment could be much heavier in terms of execution time than implementing the HBAC model, and thus our approximation HBAC program, even heavier than a normal HBAC program, could be much lighter than executing the original IBAC program in a IBAC-supported environment.

Chapter 5

Conclusions and Future Work

In this thesis we present two main researchs. The first one is a EWPDS-based formal model for dynamic access control based information flow (IBAC). A subset of the original IBAC semantics is represented by a WPDS. The verification problem of our model and an implementation in an existing WPDS 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 the last section of Chapter 3 achieves good scalability up to 20 permissions using one variable. Future work on this research includes the optimization of our implementation and searching for a data structure more suitable for our WPDS model.

The second research we present in this thesis is an algorithm to approximate a program modeled with the IBAC model to a program modeled with the HBAC model. Inside this algorithm we describe a special algorithm for the case when a variable permissions are upgraded. As seen in the example in Chapter 4, this algorithms changes dramatically the structure of the original program in order to keep the same behavior of the original IBAC program but modeled with the HBAC. As explained in Chapter 4, keeping the same behavior means that both programs allow and forbid the same security sensitive operations. Because the structure of the output program is drastically modified, especially in terms of number of procedures, the execution time of the out-

put program could be higher compared to the original program modeled by the HBAC model. However, the main reason of this algorithm is to achieve an output program that is lighter in terms of execution time than the original program modeled by the IBAC. Because a IBAC-supported environment does not exist, this matter cannot be proved, but looking at the results explained in the discussion in the last section of Chapter 4, it could be guessed that the IBAC model would be very expensive compare to the HBAC model to directly implement the IBAC in a programming environment. Future work includes the improvement of the algorithm for applying a larger subset of IBAC programs.

Moreover, another future work regarding to this second research is related with another aproach to achieve the same objective. In the paper [6] the authors propose a method for, given a recursive program and a security specification, automatically put check statements for the HBAC access control into that program. Using this same methodology, the idea is first, using a model-checking algorithm, to search in an IBAC program which check statements succeed and which ones fail. After performing this we can discover which execution paths in the IBAC program are valid (all the check statements succeed) or invalid (at least one check statement fails). The second step is, from the same program modeled this time with the HBAC and without check statements, to introduce check statements that fails in the same paths that failed in the original IBAC program. With this, we can prevent the execution of the same security-sensitive operations that are prevented in the original IBAC program. Therefore the program maintains its safety by using the HBAC model instead. However, the theoritically counterpart of this technique is the "oversafety" that produces. This means that the introduced check statements may cut a path that in the original program was a valid path. Therefore, this technique maintains the same safety of the original program but at the cost of reducing the number of valid execution paths.

Acknowledgement

This thesis dissertation would not have been possible without the support and help of several persons who contributed in some way in the preparation of this study.

First of all, I wish to express my deepest gratitude to my advisor, Prof. Dr. Yoshiaki Takata whose support, guidance and confidence in my abilities were fundamental throughout these years in the completion of this thesis.

My gratitude are also due to the members of the supervisory comimittee, Prof. Dr. Akio Sakamoto, Prof. Dr. Makoto Iwata, Prof. Dr. Kazutoshi Yokoyama, Prof. Dr. Kiminori Matsuzaki.

Thanks also to Prof. Lawrie Hunter, who not only taught me Formal Academic English but also shared with me long and interesting conversations about any topic.

My gratitude also to Prof. Dr. Shinichi Yamagiwa for all things you taugth me and did for me. Without you I would not have come to Kochi University of Technology.

Thanks to Mr. Nicholas Kidd for his useful advice and help regarding the usage of the WALi tool.

I would also like to thank to the members of the International Relationship Division; former members Kimiko Ban, Motoi Yoshida, Kimi Kiyooka and current members Prof. Dr. Akimitsu Hatta, Sonoko Fukudome, Kubo sensei, Kimiko Sakamoto, Mari Yamazaki and Rika Fujii. Thank you all for making this university such a comfortable place for the foreign students.

Thanks to all the lab members at the Takata Laboratory I have met during these years for your friendly attitude and for making this laboratory such an enjoyable environment.

Special thanks to my friend and my japanese father Shane-san. Thanks for driv-

ing me a lot of times to the airport, for making me laugh every morning in Kuzume dormitory, for helping us to find a house for me and Jingyun. Thanks for being there everytime we needed.

Thanks to my family for being there every day of these years, for believing in me and for sending me love and support from the distance.

And last but not least, I want to sincerely show my gratitude to the love of my life and the woman who has been supporting me and taking care of me every single second during the whole PhD. Thanks for giving me confidence when I needed and thanks for making my life more beautiful. Not a single sentence of this dissertation would have been possible without her.

List of Publications

Journal Papers

(1) Pablo Lamilla Alvarez and Yoshiaki Takata, A Formal Verification of a Subset of Information-Based Access Control Based on Extended Weighted Pushdown System, IEICE Transactions on Information and Systems. (Accepted, pending publication)

(2) Pablo Lamilla Alvarez, Shinichi Yamagiwa, Masahiro Arai and Koichi Wada, A uniform Platform to Support Multigenerational GPUs for High Performance Stream-based Computing, International Journal of Networking and Computing, Vol. 1, July 2011.

Conference Papers

(1) Pablo Lamilla Alvarez and Yoshiaki Takata. An HBAC-based approximation for IBAC programs. In Proceedings of the 6th International Conference on Security of Information and Networks (SIN '13), Aksaray, Turkey, 2013. ACM, New York, NY, USA, 277-281.

(2) Pablo Lamilla Alvarez, Shinichi Yamagiwa, Masahiro Arai and Koichi Wada, Elimination Techniques of Redundant Data Transfers among GPUs and CPU on Recursive Stream-based Applications, IPDPS/APDCM11, Anchorage USA, May 2011.

References

[1] M. Abadi, C. Fournet. Access Control Based on Execution History. In *11th Network and Distributed System Security Symposium*, February 2003.

[2] M. Pistoia, A. Banerjee, D.A. Naumann. Beyond Stack Inspection: A Unified Access-Control and Information-Flow Security Model. In *Security and Privacy IEEE Symposium*, pages 149–163, May 2007.

[3] L. Gong, M. Mueller, H. Prafullchandra, R. Schemers. Going Beyond The Sandbox: An Overview of the New Security Architecture in the Java^{TM} Development Kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, pages 103–112, 1997

[4] A. Banerjee, D. A. Naumann. History-based Access Control and Secure Information Flow. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 27–48, 2004.

[5] J. Wang, Y. Takata, and H. Seki. HBAC: A Model for History-Based Access Control and Its Model Checking. In 11th ESORICS, LNCS, vol.4189, pp.263–278, 2006.

[6] Y. Takata, and H. Seki. Automatic Generation of History-Based Access Control from Information Flow Specification. In 10th ATVA, pp.259–275, 2010.

[7] A. Lal, T. Reps, and G. Balakrishnan. Extended Weighted Pushdown Systems. In CAV05: Proceedings of the 17th International Conference on Computer Aided Verification, pp.434–448, 2005.

[8] S. Schwoon. Model-checking Pushdown Systems. PhD thesis, Technical University of Munich, 2002.

[9] F. Stajano and R. Anderson. The Resurrecting Duckling: Security Issues in Ad-Hoc Wireless Networks. In Security Protocols, 7th International Workshop Proceedings, LNCS, vol.1796, 1999. URL http://www.cl.cam.ac.uk/ fms27/duckling/.

[10] 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.

[11] J.Lind-Nielsen. BuDDy – A Binary Decision Diagram Package. http://sourceforge.net/projects/buddy, 2004.

[12] Y.He, Multicore-enabling a Binary Decision Diagram algorithm. May 2009.

[13] T. Reps, S. Schwoon, S. Jha, and D. Melski. Weighted Pushdown Systems and Their Application to Interprocedural Dataflow Analysis. Sci. Comput. Program., 58(1-2):206–263, 2005.

[14] F. Martinelli, P. Mori. Enhancing Java Security with History Based Access Control. In *Foundations of Security Analysis and Design*, pages 135–159, 2007.

[15] G. Edjlali, A. Acharya, V. Chaudhary. History-based Access Control for Mobile Code. In *Computer and Communicacions Security*, ACM, New York, USA, pages 38-48, 1998.

[16] K. Krukow, M. Nielsen, V. Sassone. A Logical Framework for History-based Access Control and Reputation Systems. In *Journal of Computer Security*, 16(1):63-101, January 2008.

[17] C. Sun, L. Tang and Z. Chen, Secure Information Flow by Model Checking Pushdown System, in Proceedings of UIC-ATC09, pp.586–591, 2009.

[18] S. Jha, S. Schwoon, H. Wang and T. Reps, Weighted Pushdown Systems and Trust-Management Systems, TACAS06, LNCS, vol.3920, pp.1–26, 2006. Springer-Verlag.

Appendix A

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, $(e, e) \in MOVP$ (n_0, n_1) for $e = \mathit{proj}(s, D) = \mathit{proj}(s', D)$.

(Induction step) Assume that $l > 0$. This implies that $S = n_0 \colon C; S'$. 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 $(e'', e') \in MOVP(n_2, n_1)$ where $n_2 = \mathit{head}(S')$, $n_1 = \mathit{last}(S') = \mathit{last}(S)$, $e'' = \mathit{proj}(s'', D)$, and $e' = \mathit{proj}(s', D)$. On the other hand, for each form of C we can show that $n_0 \Rightarrow^* n_2$ and $(e, e'') \in g_1(id, MOVP(n_0, n_2))$ for the conditional clause, $(e, e'') \in g_2(id, MOVP(n_0, n_2))$ for the procedure call, or $(e, e'') \in MOVP(n_0, n_2)$ for the other commands. We conclude that $n_0 \Rightarrow^* n_1$ and $(e, e') \in MOVP(n_0, n_1)$.

(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 $R = SP(n_0) \cap (\bigcap_{y \in V(E)} \mathit{proj}(s(y)))$. On the other hand, by the definition of W_π, $n_0 \Rightarrow n_2$ and $(e, e_2) \in MOVP(n_0, n_2)$ 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 $(e, e'') \in MOVP(n_0, n_2)$ when $e = \mathit{proj}(s, D)$ and $e'' = \mathit{proj}(s'', D)$. Since $MOVP(n_0, n_1) \supseteq MOVP(n_0, n_2) \otimes$ $MOVP(n_2, n_1)$, $n_0 \Rightarrow^* n_1$ and $(e, e') \in MOVP(n_0, n_1)$ where $e = \mathit{proj}(s, D)$ 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 $(e_3, e_4) \in MOVP$ (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 $(e, e_3) \in MOVP(n_0, n_3 n_2)$ for any $e_0 \in Env$ and $e_3 = e[dp \mapsto (e(dp) \cup R) \cap SP(p)]$. Therefore $(e, e_3) \in MOVP$ $(n_0, n_3 n_2)$ 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 $MOVP(n_3 n_2, n_2) = MOVP(n_3 n_2, n_4 n_2) = MOVP(n_3, n_4)$, and thus $(e, e_4) \in$ $MOVP(n_0, n_2) = MOVP(n_0, n_3 n_2) \otimes MOVP(n_3 n_2, n_2)$ where $e = proj(s, D)$ and $e_4 = proj(s'', D')$. By the definition of W_π, $MOVP(n_0, n_1) = g_2(id, MOVP(n_0, n_2)) \otimes$ $MOVP(n_2, n_1)$ and $(e, e'') \in g_2(id, MOVP(n_0, n_2))$ where $e'' = proj(s'', D)$ since g_2 forces $e''(dp) = e(dp)$. It concludes that $n_0 \Rightarrow^* n_1$ and $(e, e') \in MOVP(n_0, n_1)$.

(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 $W = write\_oracle(S_2)$. By the inductive hypothesis, $n_3 \Rightarrow^* n_4$ and $(e_3, e_4) \in MOVP(n_3, n_4)$ 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 $(e, e_3) \in MOVP(n_0, n_3 n_2)$ 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 $(e, e_3) \in MOVP(n_0, n_3 n_2)$ when $e = proj(s, D)$ and $e_3 = proj(s_0, D)$. Moreover, $n_4 \Rightarrow \epsilon$ and $(e_4, e_5) \in MOVP(n_4, \epsilon)$ 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 $(e_4, e_5) \in MOVP$ (n_4, ϵ) when $e_4 = proj(s_1, D)$ and $e_5 = proj(s_2, D)$. $MOVP(n_0, n_1) \supseteq g_1(id, MOVP$ $(n_0, n_2)) \otimes MOVP(n_2, n_1)$ and $(e, e'') \in g_1(id, MOVP(n_0, n_2))$ where $e'' = proj(s'', D)$ since g_1 forces $e''(pc) = e(pc)$. It concludes that $n_0 \Rightarrow^* n_1$ and $(e, e') \in MOVP(n_0, n_1)$.

(D) Consider the case that $C = \mathit{test}\ R\ \mathit{then}\ S_1\ \mathit{else}\ S_2$. 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 $(e, e'') \in MOVP$ (n_3, n_4) where $n_3 = \mathit{head}(S_1)$, $n_4 = \mathit{last}(S_1)$, $e = \mathit{proj}(s, D)$, and $e'' = \mathit{proj}(s'', D)$. By the definition of W_π, $n_0 \Rightarrow n_3\, n_2$ and $(e, e) \in MOVP(n_0, n_3 n_2)$ for $e = \mathit{proj}(s, D)$ since $R \subseteq D = e(\mathit{dp})$. Moreover, $n_4 \Rightarrow \epsilon$ and $(e'', e'') \in MOVP(n_4, \epsilon) = id$, and thus $n_0 \Rightarrow^* n_2$ and $(e, e'') \in MOVP(n_0, n_2)$.

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

□

Appendix B

Code of our EWPDS Implementation in WALi

Functions we added in the file BitBinRel.cpp made by Nicholas Kidd. These functions manipulate the binary relation we use to represent the weight of the EWPDS.

```
// { (a, a'[v->1]) | (a, a') \in this }
binrel_t BinRel::SetVar(int v)
{
 check_var(v);
 bdd x = bdd_ithvar(POST(v));
 return new BinRel(bdd_exist(rel, x) & x);
   // x is a predicate that states "POST(v) must equal 1,"
   // i.e. x = { (a,a') | a'(v) = 1 }.
   // At the same time, x is the variable set that consists of POST(v).
   // bdd_exist(rel, x)     = { (a,a') | (a,a'') \in this
   //                                and a'(u) = a''(u) for each u != v }.
   // Thus, bdd_exist(...) & x = { (a,a') | (a,a'') \in this
   //                                and a'(u) = a''(u) for each u != v,
   //                                and a'(v) = 1 }.
}

// { (a, a'[v->0]) | (a, a') \in this }
binrel_t BinRel::ClearVar(int v)
```

```
{
 check_var(v);
 bdd x = bdd_ithvar(POST(v));
 return new BinRel(bdd_exist(rel, x) & bdd_not(x));

   // bdd_not(x) is a predicate that states "POST(v) must equal 0,"
   // i.e. bdd_not(x) = { (a,a') | a'(v) = 0 }.
}

// { (a, a'[to->a(from)]) | (a, a') \in this }
binrel_t BinRel::SubstituteVar(int from, int to)
{
 check_var(from);
 check_var(to);
 bdd x = bdd_ithvar(POST(to));
 bdd y = bdd_ithvar(PRE(from));
 return new BinRel(bdd_exist(rel, x) & bdd_biimp(x, y));

   // bdd_biimp(x,y) is a predicate that states "POST(to)=1 iff PRE(from)=1,"
   // i.e. bdd_biimp(x,y) = { (a,a') | a'(to) = a(from) }.
}

// { (a, a'[to->a(to)&a(from)]) | (a, a') \in this }
binrel_t BinRel::AndVar(int from, int to)
{
 check_var(from);
 check_var(to);
 bdd x = bdd_ithvar(POST(to));
 bdd y = bdd_ithvar(PRE(from)) & bdd_ithvar(PRE(to));
 return new BinRel(bdd_exist(rel, x) & bdd_biimp(x, y));
}
```

```
// { (a, a'[to->a(to)|a(from)]) | (a, a') \in this }
binrel_t BinRel::OrVar(int from, int to)
{
 check_var(from);
 check_var(to);
 bdd x = bdd_ithvar(POST(to));
 bdd y = bdd_ithvar(PRE(from)) | bdd_ithvar(PRE(to));
 return new BinRel(bdd_exist(rel, x) & bdd_biimp(x, y));
}

//TestVar is used for the test operation.
binrel_t BinRel::TestVar(int v)
{
 check_var(v);
 bdd x = bdd_ithvar(PRE(v)) & bdd_ithvar(POST(v));
 return new BinRel(bdd_exist(rel, x) & x);
}

//ComposeVar is the operation to compose the weight for the merge function
binrel_t BinRel::ComposeVar(binrel_t w1, int from, int to, int max){
 check_var(from);
 check_var(to);
 check_var(max-1);
 bdd x = bdd_ithvar(POST(from));
 for(int i = from + 1; i < to; i++){
    x = x & bdd_ithvar(POST(i));
 }
  bdd y = bdd_ithvar(POST(0));
 for(int j = 1; j<from; j++){
   y = y & bdd_ithvar(POST(j));
 }
   for(int z = to; z<max; z++){
```

```
    y = y & bdd_ithvar(POST(z));
  }

  return new BinRel(bdd_exist(rel, x) & bdd_exist(w1->rel, y));
}
```

Code of the Resurrecting Duckling example. Because the rest of the examples follow a very similar structure we selected this program to as representative of all the examples.

```
using namespace wali::domains::binrel;

/*
Example program to be analyzed
int x;
void main() {
 imprintA() or imprintB(); 0
 unprintA() or unprintB(); 1              Permissions A,B
 return;                2
}
void imprintA(){
check(x,AB);       3
x= 1;              4      Permission A
return;            5
}
void imprintB(){
check(x,AB);       6
x= 1;              7      Permission B
return;            8
}
```

```
void unprintA(){
check(x,A);       9
x= 1;              10      Permissions A,B
return;            11
}
void unprintB(){
check(x,B);       12
x= 1;              13    Permissions A,B
return;            14
}

Permissions availables {xA,xB}

Binary relation {xA,xB,dpA,dpB,pcA,pcB}
*/
int doReach()
{
 using wali::Key;
 //using wali::wpds::WPDS;
 using wali::wpds::ewpds::EWPDS;
 using wali::wfa::WFA;
 using wali::MyMergeFn;

 std::ofstream prf("output.txt");

 struct timeval start, end;
 long mtime, seconds, useconds;

 int nPermissions = 2; //Number of permissions availables
 BinRel::initialize(nPermissions*3);

 EWPDS myWpds;
```

```
Key p = wali::getKey("p");
Key accept = wali::getKey("accept");
Key n[3+nPermissions*6];
int i;
for( i=0 ; i <= (3+nPermissions*6) ; i++ ) {
  std::stringstream ss;
  ss << "n" << i;
  n[i] = wali::getKey( ss.str() );
}
binrel_t ID = BinRel::Id(); //ID weight
binrel_t InprTestW = ID; //Weight of the check operations at the imprint
    functions
binrel_t AssigW = ID; //Weight of the assignment operations

for ( i = 0; i < nPermissions ; i++){

  binrel_t CallWeight = ID->SetVar(nPermissions+i); //Weight for the
      CallInprintX. Set 1 the component i of the dynamic permission. In the
      first iteration would be the component A because we are calling the
      imprintA()
  InprTestW = InprTestW->TestVar(i);              //We create the weight here
      and it will be used later
  AssigW = AssigW->SubstituteVar(nPermissions+i,i); //We create the weight here
       and it will be used later

  for (int j = 0; j < nPermissions; j++){
    if(i!=j){
    CallWeight = CallWeight->ClearVar(nPermissions+j); //Set to 0 the rest of
        the components of the dynamic permission. If we are calling imprintA(),
         this will set to 0 the rest of the components (B,C,D...)
    }
  }
```

```
   MyMergeFn* mf = new MyMergeFn(CallWeight);
  myWpds.add_rule( p, n[0], p, n[3+i*3] ,n[1],CallWeight,mf);   //Add the rules
       of calling the imprint functions from imprintA to the last
      imprintnPermissions
  MyMergeFn* mf2 = new MyMergeFn(ID);
  myWpds.add_rule( p, n[1], p, n[3+nPermissions*3+i*3] ,n[2],ID,mf2); //Add the
       rules of calling the unprint functions from unprintA to the last
      unprintnPermissions. Here the dynamic permissions dont change

}

for (i = 0; i < nPermissions; i++){
  myWpds.add_rule( p, n[3+i*3], p, n[4+i*3], InprTestW);
                                    //Rules for the Check Operations at the
      imprint functions
  myWpds.add_rule( p, n[4+i*3], p, n[5+i*3], AssigW);
                                       //Rules for the assignment Operations
      at the imprint functions
  myWpds.add_rule( p, n[5+i*3], p, ID);
                                                  //Rules for the returns
      at the imprint functions
  myWpds.add_rule( p, n[3+nPermissions*3+i*3], p, n[4+nPermissions*3+i*3],
      ID->TestVar(i)); //Rules for the check operations at the unprint
      functions
  myWpds.add_rule( p, n[4+nPermissions*3+i*3], p, n[5+nPermissions*3+i*3],
      AssigW);     //Rules for the assignment Operations at the unprint
      functions
  myWpds.add_rule( p, n[5+nPermissions*3+i*3], p, ID);
                                       //Rules for the returns at the unprint
      functions
  printf("Iteration2for %d done\n",i);
}
```

```
myWpds.add_rule( p, n[2], p, ID);
                                                    //Rule for return at
    the main function

// Perform poststar query
WFA query;
/*std::cerr*/ prf << "\t> Adding p,n0,acc to query...";
std::cerr<< "\t> Adding p,n0,acc to query...";
query.addTrans( p, n[0], accept, BinRel::Id());
 /*std::cerr*/ prf << "> done\n";
std::cerr<< "> done\n";
query.set_initial_state( p );
query.add_final_state( accept);
WFA answer;
gettimeofday(&start,NULL);
myWpds.prestar(query,answer);
gettimeofday(&end,NULL);

seconds = end.tv_sec - start.tv_sec;
useconds = end.tv_usec - start.tv_usec;


printf("PostStar execution time: %ld, %ld, %ld, %ld microseconds\n",
    end.tv_sec, start.tv_sec, end.tv_usec, start.tv_usec);

printf("PostStar execution time: %.2f
    milliseconds\n",(float)((float)(end.tv_sec - start.tv_sec)*1000 +
    (float)(end.tv_usec-start.tv_usec)/1000));

return 0;
} /** end of main **/
```

```
int main()
{
 doReach();
 //while(1);
 std::cerr << "# Trans : " << wali::wfa::Trans::numTrans << std::endl;
 std::cerr << "# States : " << wali::wfa::State::numStates << std::endl;
 std::cerr << "# Rules : " << wali::wpds::Rule::numRules << std::endl;
 std::cerr << "# Configs : " << wali::wpds::Config::numConfigs << std::endl;
 std::cerr << "# Variables : " << BinRel::getNumVars() << std::endl;
 //while(1);
 return 0;
}
```