論 文 内 容 の 要 旨

1. Introduction

Access control models are adopted in modern component-based systems, specifically the Stack-Based Access Control (SBAC) supported in environments such as Java virtual machines and Microsoft .NET Common Language Runtime (CLR). However SBAC may allow *non-trusted code* to influence a security-sensitive operation performed by *trusted code* because SBAC does not retain the security information of previously executed methods. To solve this problem, Abadi et al.[1] introduced a novel approach called History-Based Access Control (HBAC) which registers the history of all previously executed methods. HBAC systems ensure that all the code previously executed is sufficiently authorized to access a protected resource. However, since not all the code previously executed may have influence on the protected resource, the HBAC is excessively restrictive in some cases. Pistoia et al. formally presented in [2] a novel security model called Information-Based Access Control (IBAC) which has proven to be less restrictive than HBAC while maintaining the level of security assurance of HBAC. The IBAC model tracks not only the permissions of blocks of code but also the dynamic permissions of each variable. Therefore, the IBAC verifies that only the code responsible for a security-sensitive operation is sufficiently authorized. Since the model proposed in [2] is not aimed at formal verification, the model is not applicable to model checking problems. This thesis proposes a formal model for IBAC programs suitable for formal verification. We model the behavior of an IBAC program as a weighted-pushdown system (WPDS) [4]. This extension of the transition system called Push-down System (PDS) naturally models the behavior of an IBAC program in which subsets of permissions are maintained and altered at every procedure call and return. Moreover, WPDS improves the efficiency of model-checking compared with a modeling by PDS where the subsets of permissions are encoded into stack symbols. In this thesis we also address the problem of implementing the IBAC model in a real environment. Even though the HBAC and the IBAC models are theoretically safer than SBAC, only the latest is really implemented in a programming environment. This thesis 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. Note that a few works exist

regarding HBAC implementations in real environments [6] [7]. Our approximation method could be integrated with these HBAC implementations that already exists, and therefore achieve a runtime for IBAC programs without directly implementing the IBAC model.
In this research we present a WPDS-based formal model for dynamic access control based on 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. The verification problem for HBAC has been discussed in works such as [3] but no formal verification has been proposed for IBAC. Therefore, our work is the first one that discusses and implements the model-checking problem for IBAC programs. As well, we propose an algorithm to approximate an IBAC program to an HBAC program in order to be able to execute an IBAC program into an environment that supports the HBAC model. This way, we could achieve a runtime for IBAC programs taking advantage of the already implemented HBAC runtimes without explicitly implementing a runtime for IBAC programs.
Summarizing, the results and contributions of this thesis are mainly two:

- A verification model and its implementation in an existing model-checking tool for a subset of IBAC programs.
- An algorithm to approximate a subset of IBAC programs to HBAC programs. With this we implicitly achieve a runtime for that subset of IBAC programs because they could be executed as HBAC programs in an already implemented runtime for the HBAC model.

2. IBAC Program

The syntax and the semantics of an IBAC program are formally defined in [2]. Like previous access control models (SBAC, HBAC), a set of permissions is statically assigned to each procedure and the runtime system dynamically maintains the current permissions of the execution process based on the static permissions of each procedure called. When a procedure is called, the current permissions of the process are intersected with the static permissions of the procedure. In SBAC the process recovers the previous current permissions when the called procedure finishes. On the other hand, in HBAC the current permissions are maintained when the called procedure finishes. In both SBAC and HBAC, a permission-check command **test** R **then** $S1$ **else** $S2$ is statically placed by the programmer just before each security-sensitive operation. At the permission-check command it is tested

whether the current permissions includes as a subset the set of permissions specified in the check command. The negative case of this check is treated as a security violation. In the IBAC model case, the current set of permissions is dynamically maintained not only for the process but also for variables. Another kind of permission-check command **test** *R* **for** *x* is used for testing the current permissions of variable *x*.

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

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

BadFunction(){
return;
}
```

Example a

```
int main(){
BadFunction();
write x;
return;}
```

Example b

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

Fig. 1. Program Examples of IBAC and HBAC.

Example 1. Figure 1 shows a small program that illustrates the advantage of IBAC over HBAC. In this program, the trusted function *main* calls the non-trusted function *BadFunction*, and then the function *main* tries to modify the file *password.txt*. In the version *a* of this program, the *password.txt* file is requested and returned by *BadFunction*, which does not possess any permission over the file and thus the main function would not be able to modify the file. In this example, HBAC would detect this security violation by checking the dynamic permissions at the sensitive operation *write(x)*. Since these dynamic permissions are at most the static permissions of *BadFunction*, the operation is not executed because *BadFunction* does not have sufficient permission. IBAC also detects this violation by checking the permissions associated with variable *x*. Since this variable was modified at *BadFunction*, the operation is not executed because the permissions associated to variable *x* are the same as the static permissions of *BadFunction*. On the other hand, in the version *b* of this program, the *BadFunction* does not have any influence on the *password.txt* file. Therefore the operation *write(x)* should be allowed to be executed. However, HBAC also negates the execution of the sensitive operation because, since *BadFunction* is called, the dynamic permissions are still intersected with the dynamic

permissions of *BadFunction*. As a result, the behavior of HBAC is exactly the same in this example as in example *a*. IBAC on the contrary allows the sensitive operation to be executed in example *b* because the variable *x* is modified in the function *main,* not in *BadFunction.* As a result, the permissions associated to *x* are the static permissions of *main* which are enough to execute the sensitive operation *write(x)*. This example illustrates how HBAC is more restrictive than IBAC. After explaining the IBAC model and its characteristics, we present our research objective in the next section.

3. A Formal Verification Model

Our objective in this part of the research is to define a model suitable for model-checking for IBAC programs. In this section model-checking technique and the model chosen for representing IBAC programs are presented.

3.1.Model Checking

Model-checking is a technique used both in software and hardware systems that verifies some property of a given system. For example, one could verify the absence of deadlocks and other critical states that causes the given system to crash. The main advantage of this technique is that it automatically explores all the states of the system, therefore testing all the possible execution paths. This way, we can completely ensure that the property we wanted to verify, i.e. deadlock, never occurs.

Applying model-checking technique in IBAC programs, we could verify that no security violation happens in a given IBAC program. For example, a bug in a test command when coding an IBAC program could cause an unauthorized user to access some files or operations beyond his permissions. This wrong behavior can be detected by analyzing the program with a model-checking tool. However, before applying model-checking, we need to model the system we want to verify using a mathematical model suitable for model-checking techniques. The model chosen for our case is called Pushdown System.

3.2.Pushdown Systems

Pushdown Systems (PDS) [5] are a stack-based transition systems that provide a natural execution model in the analysis of sequential programs. A PDS is composed by a finite set of control locations and a stack. This stack is unbounded, i.e. does not have a limit, and works the same way a normal stack structure works: elements can be pushed or popped from the stack in a Last-In-First-Out (LIFO) order. The stack structure provides a natural representation of procedure calls, which are the most important part in IBAC programs,

and recursion because the order of which the functions are called is always kept by the stack. This way the PDS can correctly return to the position of the callee function after the called function finished. The stack of a PDS would be enough to represent the information we need from an IBAC program. This is, the flow (execution path) of the program and the set of permissions of each variable. However, we use an extension of PDS called Weighted PDS (WPDS) [4]. This extension adds to the PDS a weight that can represent finite and infinite data domains. Even though our data domain (permissions of each variable) is finite, WPDS model-checking is more optimized for finite data domains than normal PDS model-checking. Therefore using WPDS we can achieve better performance in the model-checking process. The weight in our WPDS model for IBAC programs is codified as in the following example: a program with 1 variable *x* and two different permissions read and write *r, w*, would be represented as (*xr, xw*) where these two elements are boolean variables. If both elements are 1, it would mean that both permissions read and write are assigned to variable *x*.

The next step after codifying an IBAC program as a WPDS model is to decide the property we want to verify using model-checking. This is explained in the next section.

3.3.Formal Verification Problem

Let us discuss the model-checking problem of our WPDS model. Given two points of a program $n0$ and n the model-checking technique calculates a weight for each execution path that can be taken between these two points. The final weight from $n0$ to n would be the combination each weight calculated for each path that can be taken from $n0$ to n. The process of calculating this weight is called Meet-Over-All-Paths (MOVP). Let us imagine the following example: if a user A performs a login in a system, and at some point after the login this user A can access files or operations that belongs to another user B, then the system violates a security property. In our model, if the MOVP between two points is empty, it means that it is impossible to reach the final node n from the initial node $n0$ because at least one permission test statements fails for all the possible paths. However, if the weight is not empty it means that it is possible to reach the final node from the initial node through some paths. In our example, this would mean that the security property we wanted to check is violated so user A can access unauthorized operations. Therefore, by checking the emptiness of the final weight of the MOVP operation, we can test any safety property in an IBAC program.

4. Approximation-Based Implementation

Our objective in this part of the research is related with the implementation of the IBAC model. As mentioned before, there are no implementations of the IBAC model in real runtimes whereas there exist for the HBAC model. Therefore, if we manage to approximate a given IBAC program to a program with HBAC semantics, that IBAC program could be implemented in a runtime that supports HBAC.

4.1.Approximation Algorithm

The objective of this algorithm is to produce a HBAC program that has the same behavior of a given input IBAC program. Having the same behavior means that the permission check statements produce the same output in both programs, i.e. the same sensible operations are allowed or not allowed to be executed in both programs.

In order to achieve this, the algorithm basically consists of three steps; each of these address one of the three main differences between the HBAC and the IBAC models. These differences are:

- Different number of dynamic permission sets. IBAC keeps track of a set of permissions for each variable in the program whereas HBAC does not.
- When a function finalizes in IBAC, the current dynamic permissions are restored to its previous value whereas in HBAC not.
- HBAC does not modify the current permissions when an assignment is executed whereas IBAC does modify the current permissions of the assigned variable. In this step we differentiate two cases: the variable permissions are downgraded (after the assignment the variable has the same or less permissions) or the variable permissions are upgraded (the variable has more permissions after the assignment). The algorithm first classifies all the assignments in one of these two cases and then treats them in a different way.

References

1. M. Abadi, C. Fournet. Access Control Based on Execution History. In Proceedings of the 11th Network and Distributed System Security Symposium (NDSS 2003), Feb.2003.
2. M. Pistoia, A. Banerjee, and D.A. Naumann. Beyond Stack Inspection: A Unified Access-Control and Information-Flow Security Model, In Security and Privacy, 2007. SP '07. IEEE Symposium on, pp.149--163, May 2007.

3. J. Wang, Y. Takata, and H. Seki. HBAC: A Model for History-Based Access Control and Its Model Checking. In 11th ESORICS, LNCS, vol.41--89, pp.263278, 2006.
4. 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.
5. S. Schwoon. Model-checking Pushdown Systems. PhD thesis, Technical University of Munich, 2002.
6. G. Edjlali, A. Acharya, V. Chaudhary. History-based Access Control for Mobile Code. In Computer and Communications Security, ACM, New York, USA, pp 38--48, 1998.
7. 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.