Model Checking Security Policy Model Using Both UML Static and Dynamic Diagrams

Liang Cheng
State Key Laboratory of Information Security
Institute of Software, Chinese Academy of Sciences
Beijing 100190
chengliang@is.iscas.ac.cn

Yang Zhang
State Key Laboratory of Information Security
Institute of Software, Chinese Academy of Sciences
Beijing 100190
zhangyang@is.iscas.ac.cn

ABSTRACT

An operating system relies heavily on its security model to defend against malicious attacks. It has been one of the hottest research domains for decades to validate security models' correctness by formal methods during the development of security operating systems. However, current studies on the formal verification of security models are sometimes too sophisticated for the developers of operating systems, who are usually not experts in mathematical reasoning and proving. So representing a security model in UML becomes a compromise choice for the developers' verification work during system developing. In this paper, we propose a new method to verify the security policy model against the security goals using model checker SPIN and UML modeling language. Given a security policy model and the security property to be validated, our approach leverages UML class diagrams and statechart diagrams to specify its state model and its state transitions respectively. Then we translate these UML diagrams into the input language of SPIN automatically, as well as the security property. The conformance between the security goal and security model can finally be analyzed by SPIN. We proved the effectiveness of our approach by checking the violation of confidentiality of the DBLP model.

Categories and Subject Descriptors

D.4.6 [**OPERATING SYSTEMS**]: Security and Protection—*Verification*; D.2.4 [**SOFTWARE ENGINEERING**]: Software/Program Verification—*Model checking*

General Terms

Security, Verification

Keywords

Security Model; Formal Verification; Model Checking; UML

1. INTRODUCTION

Security models play an important part in ensuring the security of the operating systems. A tiny flaw in the models may provide opportunities for malicious attackers to gain extra privileges or plant a trojan horse. To validate the correctness of the security policy models, formal methods and tools are the most popular techniques involved [1, 2, 3, 4, 5]. However, it is also a common practice that the verifiers of an operating system are as well the designer of verification tools. The developers of an operating system are usually considered not so capable to verify their own system independently, for the sophisticated mathematical foundation needed during the verification procedure.

On the other hand, the Unified Modeling Language (UML) [6] is nowadays the de-facto standard modeling language and widely used in software/OS development. The usage of UML for the specification of security aspects is an attractive aim, since system engineers are used to the UML notation and the accompanying tools[7]. So it is intuitive to specify security models by UML and facilitate the developers of operating systems to verify their model's correctness at development time, without extra effort to learn the boring formal method theories.

The studies of leveraging UML to analyze the security model have been quite active in both academia and industry [8, 9, 10, 11, 12, 13, 14, 15, 16]. Most of existing approaches mainly differ from the kinds of UML diagrams they use to represent the security models and the back-end analysis engines to explore security models' vulnerability. UML provides class diagrams, sequence diagrams and state-machine diagrams, which can all be used for security model description. And lots of automated reasoning tools, such as model checking, theorem proving or Alloy [17], a structural modeling language based on first-order logic, have been leveraged to analyze the vulnerabilities in security models.

Most of these existing approaches focus on checking the correctness of a configuration of a security model. They consider the security model itself is flawless by default. However, when designing a security operating system, it is likely to tune traditional security models a little for the implementation reason [18]. And how to verify the soundness of the variation of a security model is out of current solutions' consideration.

In this paper, we present a new approach to validate the security model itself. A security policy model is represented as a finite state automata (FSA) in our proposal. The UML class diagrams and state-machine diagrams are used to describe the relationship between FSA's elements and the state transitions of the FSA respectively. We use the state of art model checking tool, SPIN [19] as our analysis engine. These UML diagrams are translated into SPIN's input language–PROMELA by automated tools and the property to be validated is specified in the SPIN model as assert specifications. The counterexamples, which means the deficiency in the model, will be simulated when the assert fails. Our contributions are as follows:

1. We propose a new approach to specify security model in UML language. Given a FSA of a security model, our approach can describe the static and dynamic properties of the security model simultaneously.
2. We study the satisfiability of confidentiality of the DBLP model by our approach. Our experiment showed that the DBLP model fails to protect the secrecy of a system. And a violation path was also generated.

The rest of the paper discusses our approach with more detail. Section 2 discusses some basics of the UML model and security model. In section 3, we describe in detail how to use UML diagrams to specify the security policy model, how to translate the UML models into SPIN's input language and how to model check the correctness of the security policy model. Next, we take the DBLP [18] model as the example to demonstrate how our approach works and provides some discussions of the result of it in Section 4. We also give a comparison to related works in Section 5 and close with an outlook of our future work in Section 6.

2. PRELIMINARIES

Generally, a security model can be described as a finite state automata(FSA): $M = (V, V_0, L, T, F, C)$, in which

- V is a finite set of states;
- V_0 is a set of initial states, and $V_0 \subseteq V$;
- L is a finite set of labels;
- T is a set of state transitions, and $T \subseteq (V \times L \times V)$;
- F is a set of final states, and $F \subseteq V$
- C is the constraints of V and T, which is equivalent to security policies.

If the initial states of the state machine are safe on certain property, and every state derived by the transition rules or constraints system is always "safe" on that property, we can conclude that the security policy model is safe on that property. That is, given a security model M and a security property (goal) P , if

$$(V, C) \models P, \text{and } (V_0, C) \models P$$

hold, we call the security model M *satisfies* the security goal P.

UML modeling language [6] also has *statechart diagrams* to specify state machines, in which nodes correspond to states and directed arcs correspond to *state transitions* labeled with *event* names. A statechart is usually attached to a class and describes the response of an instance of the class to events that it receives. The class is usually specified by a class diagram. In general, a state transition has an event *trigger*, a *guard condition* and an *effect*. The trigger specifies the event that enables a transition, and the effect is an action or activity that is executed when a transition fires. A guard condition, however, is a boolean expression. If it is true, the transition fires, or the transition fails.

So intuitively, the security state sets of V, V_0 and F in the definition above can be modeled by UML class diagrams. The set of security state transitions T can be represented by the corresponding statechart diagrams. And the label set L and the constraint set C can be translated into event triggers and guard conditions of state transitions respectively.

3. DESIGN OF OUR APPROACH

Our verification method can be illustrated as Fig.1. It has three steps. First, we specify the security policy model, including its definitions and security rules, in the form of UML diagrams. Second, after we get the UML description of the security policy model, we need a kind of abstract syntax to translate the UML diagrams into the input language of a model checker. And then, the model checker takes the descriptions obtained by the second step and the security property need to demonstrate as inputs, and traverses all states in the system to find if there is a violation of the property. The security property, as the other input of the model checker, should also be represented in the form that can be accepted by the model checker. If the model checker finds a counter-example of the property, it proves that the security policy model does not conform to the security property. And we can get the location where the violation happens according to the counter-example. If there is not any counter-example, the consistency of the model and the property can be demonstrated.

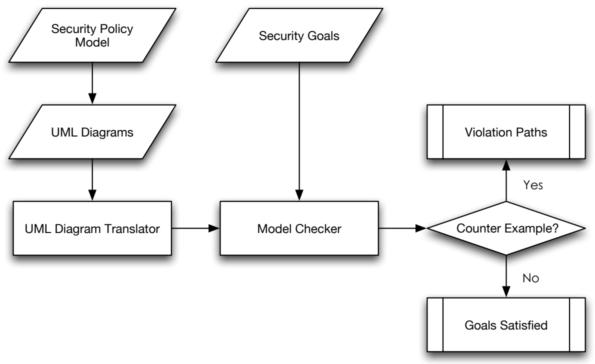

Figure 1: Security Policy Model Verification Framework

3.1 Representing a security policy model in UML

In security policy models, a system's security states are denoted by subjects, objects and access modes in common. Typically, BLP model describes a system state as $v \in V, V = (b, M, f)$, in which

- $b \subseteq (S \times O \times A)$ includes all access requests that subjects submit for certain objects;
- S is the set of all subjects in that system;
- O is the set of all objects in that sysem;

- $A = \{read, write, append, execute\}$ is the set of access modes;
- M represents the access control matrix (ACM);
- $f = (level(s), level_c(s), level(o))$, represents the sensitivity label function of $s \in S$ and $o \in O$.

However, different from the ordinary system behavior models that verified by model checkers, security policy models are highly abstracted. The cardinal number of its state set, access control matrix are not settled. Nevertheless, model checkers only accept a model that has determinate number of states, otherwise the problem of state explosion occurs[10]. Likewise, a UML statechart diagram is also based on the instance of corresponding class diagram. So before we represent the security model by UML diagrams, we need first to determine the number of system states, the size of the access control matrix and other abstract structure's contents.

Furthermore, there is another problem at this step. As mentioned above, the b in a state of a security model (SoSM) v includes all access requests between subjects and objects. It is impractical to map all these states to the states of a UML statechart diagram (SoUD) directly. Alternatively, we represent the statechart of the security model from the view of access mediation. That is, the SoUDs denote if an access request is approved or not, and the transition of a SoSM is represented by the activity or action of the SoUD.

To traverse all SoSMs, we leverage two-layered loops. Given a subject, the inner loop traverses all the objects and makes all kinds of access requests, if a request is authorized by the security policies, the system is then updated. The outer loop, accordingly, traverses all the subjects in the security model. The algorithm is illustrated as Fig. 2.

```
1  foreach  $s \in S$ do
2      Initialize the set of accessed (s) and info (s);
3      foreach $o \in O$ do
4          a = ChooseAccessMode ();
5          while $a$ is new do
6              Check if (s, o, a) is allowed by security rules;
7              Update accessed (s) and info (s);
8              a = ChooseAccessMode ();
9          end
10     end
11     Check if there is any violation in accessed (s) based
       on info (s);
12     Report the violation if it exists;
13 end
```

Figure 2: Algorithm for System State Simulation

To simulate the transitions of SoSMs, we define two UML classes. One takes the charge of sending the subject's access as a client, and the other plays the server role to determine whether the request is legal or not. Their interaction is shown as Fig.3. The client at first initializes the subject's sensitivity level. Next, the subject obtains the object to be accessed $o \in O$ and the access permission$a \in A$. The object and the permission can be generated randomly or in a certain order, which ensures that every state in the security policy model can be accessed. Then, the client submits the request triple (s, o, a) to the server. If the request is authorized, it transits the system to the next state. Whether the request is legal or not, the client submits a second request. The permission and the object to be accessed can be different for the two requests. On the other side, the server checks the request according to the security rules and access control matrix and returns its judgment to the client.

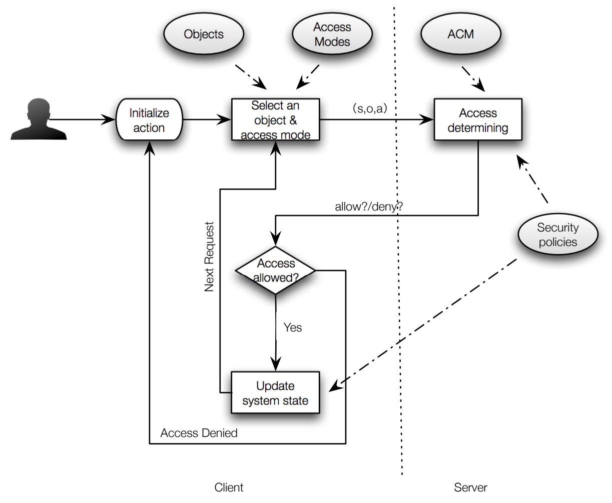

Figure 3: Modeling Security State Transition

3.2 Translating UML diagrams into a model checker's input

After representing the behaviors of the security policy model by UML statechart and class diagrams, we translate all the UML diagrams into the input of model checkers. In this step, we adopt the algorithm proposed in [20], which support PROMELA, the input language of the model checker SPIN, as its output.

As our approach only relates to statechart and class diagrams of UML, we refer to the abstract syntax of UML state machines in [20] as follows.

Definition 1. *A state s in state machines has:*

- *A kind kind(s)* $\in$ {`initial, junction, join, fork`},
- *An entry action entry(s)* $\in$ *Act,*
- *An exit action exit(s)* $\in$ *Act.*

A pseudostate is a state s with $kind(s) \in$ {`initial, junction, join, fork`}; we require that $entry(s)$ = `skip` and $exit(s)$ = `skip` for each pseudostate s. A composite state is a state s with $kind(s) \in$ {`composite, concurrent`}.

Definition 2. *A state hierarchy is given by a tree (S, E) where S is a finite set of states and $E \subseteq S \times S$ a non-empty substate relation such that the constraints below are satisfied. We write $substates(s) = \{s' \in S | (s, s') \in E\}$ for the substates of state s:*

- *if $substates(s) \neq \emptyset$, then kind(s)* $\in$ {`composite,concurrent`}
- *if kind(s)=*`concurrent`*, then #substates* ≥ 2 *and kind (s')=*`composite`*, for* $\forall s' \in substates(s)$

- *if kind(s)=*`composite`*, then #{s∈ substates(s)|kind (s') =* `initial`*}≤ 1*

Definition 3. *Given a state hierarchy $H = (S, E)$, a transition t over H has a source state source(t) $\in S$, a target statetarget(s) $\in S$, a triggering event trigger(t) $\in$ Event, a guard expression guard(t) $\in$ Exp, and an effect action $effect(t) \in Act$, such that the following constraints are satisfied:*

- *$kind(source(t)) \neq$* `final` *and $kind(target(t)) \neq$* `initial`*;*
- *if $kind(source(t)) \in \{$*`initial,fork`*$\}$, then target(t) is not a pseudostate;*
- *If kind(source(t))=*`initial`*, then container(target(t)) = container(source(t));*
- *If kind(target(t))=*`join`*, then source(t) is not a pseudostate;*
- *If kind(source(t))=*`composite`*, then kind(con-tainer (source(t))) $\neq$* `concurrent`*;*
- *If kind(target(t))=*`composite`*, then kind(con-tainer (source(t))) $\neq$* `concurrent`*;*
- *If $kind(sourcr(t)) \in \{$*`initial,fork,join`*$\}$, then guard(t) =* `true`*;*
- *If kind(target(t))=*`join`*, then effect(t)=*`true`*;*
- *If kind(soucre(t))=*`initial`*, then effect(t)=*`skip`*;*
- *If source(t) is a pseudostate, then trigger(t)=**

Definition 4. *A state machine is given by (H, T), in which H=(S, E) is a state hierarchy, and T is a finite set of transitions over H, We write outgouings(s)$\{t \in T|source(t) = s\}$, $incomings(s) = \{t \in T|target(t) = s\}$,$source(M) = \{source(t)|t \in M\}$,$target(M) = \{target(t)|t \in M\}$, such that the constraints below are satisfied for $\forall t \in T$:*

- *If kind(t)=*`initial`*, then $\#outgoings(s) = 1$;*
- *If kind(t)=*`junction`*, then $\#incomings = 1$ and $\#out-goings(s) \geq 1$;*
- *If kind(t)=*`fork`*, then $\#incomings = 1$ and $\#out-goings(s) \geq 2$;*
- *If kind(t)=*`fork`*, then there is an $s' \in S$, kind(s') =* `concurrent` *such that $targets(outgoings(s)) \subseteq substates_+(s') \setminus substates(s')$ and the following holds: if $t, t' \in outgoings(s)$ such that $\{target(t), target(t') \subseteq substates_+(s'')\}$ for some $s'' \in substates_+(s')$ then t=t';*
- *If kind(s) =* `join` *then conditions (3) and (4) hold likewise with replacing target by source and outgoings by incomings.*

3.3 Model checking the security policy model

Model checkers are based on the temporal logic. It traverses all the possible behavior paths and checks whether the property to be verified is satisfied or not at each step. As for the verification of security policy models, the properties to be verified turn out to be the security goals that the model must satisfy. There are two types of temporal logic used in model checking: linear temporal logic(LTL) and computation tree logic(CTL), which are both capable of goal description. In this paper, we specify the security goal in the form of linear temporal logic for SPIN can only accept it.

We take the confidentiality property as the example. The target of the confidentiality property is to avoid the non-trusted subject leaking the information in higher sensitivity level to another subject with a lower sensitivity level. Strictly speaking, the confidentiality property allows no information flow from higher level to lower level, including the covert channels. But security policy models aims to avoid illegal explicit information flow, without considering the covert channels, so we can represent the confidentiality property in LTL formule as follows.

Definition 5. *We define*

- *$Readlike : O \times S \to \{$*`True, False`*$\}$, which determines whether subject s access object o with the permission* `ReadOnly`*. If yes, return* `True`*; or return* `False`*;*
- *$Writelike : O \times S \to \{$*`True, False`*$\}$, which determines whether subject s access object o with the permission* `WriteOnly`*. If yes, return* `True`*; or return* `False`*.*

Denote s as a non-trusted subject, $O(s)$ as the object set which s has accessed, and $level_c(o)$ as the current sensitivity level of the object o. A system can be regarded as confidential iff:

$$\neg\exists o_1, o_2 \in O(s) \bullet (Readlike(o_1) \to \Diamond Writelike(o_2)) \wedge level_c(o_2) \leq level_c(o_1) \qquad (1)$$

Which means a non-trusted subject cannot write to a lower level object after it reads from an object with a higher level.

And now, given a security goal in the form of LTL formula, and a security model in the form of PROMELA, the model checker SPIN can check the satisfiability of the formula automatically. If the formula does not hold somewhere, the model checker can generate a path which leads to the violation location. And if the formula still holds after traversing all the possible system state, the model checker will give the right result.

4. CASE STUDY: THE VERIFICATION OF DBLP

To clarify our approach in detials, we take the DBLP security policy model as an example to show how it works in this section.

4.1 DBLP introduction

DBLP was first proposed in [18]. It aims to improve the practicality of BLP [21] model by adding rules that modify non-trusted subjects' sensitivity level dynamically. DBLP inherits the traditional description of BLP model, but it modifies the sensitivity level of both subjects and objects. In DBLP, subjects have three sensitivity labels:

- f_s, the maximum sensitivity level a subject ever has,
- a_min_s, minimum writable sensitivity level,
- v_max_s, maximum readable sensitivity level.

whereas the objects have the minimum sensitivity level L_min_o and maximum sensitivity level L_max_o.

Furthermore, DBLP defines single-level objects and multi-level objects, denoted by $single(o)$ and $multi(o)$ respectively. The former can be accessed by both trusted subjects and non-trusted subjects, while the latter can merely be accessed by trusted subjects. Non-trusted subjects can access the object which $L\_min_o(o) \neq L\_max_o(o)$, only if the access permission can be determined by one sensitivity level among the range of the object's sensitivity levels. Detailed description of the DBLP can be found in [18].

4.2 DBLP in UML

We construct a typical 4-level military confidentiality model as an instance of the DBLP model. We represent the set of sensitivity levels as {1, 2, 3, 4} for the sake of convenience. The level is in ascending order, and the level 4 corresponds to the top-secret level. However, non-trusted subjects own the root permission to all objects initially.

According to the representation of the security policy model in Sect.3.1, the state variables that need to be included in the system state are:

- a_min_s: the minimum sensitivity label of the subject;
- v_max_s: the maximum sensitivity label of the subject;
- L_min_o: the minimum sensitivity label of the object;
- L_max_o: the maximum sensitivity label of the object;
- $level_c$: current sensitivity label of the object.

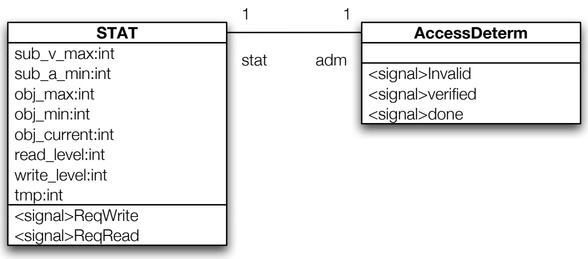

Figure 4: UML Class Diagram of DBLP

Besides, we introduce variables *read_level* and *write_level* to record the $level_c$ of objects the subject has ever accessed. The complete UML class diagram of client and server part of Fig. 3 are presented below as shown in Fig. 4. The class STAT represents the client and the class AccessDeterm is for the server.

The variable *tmp* in class STAT is a temporary variable used for the sensitivity level switch. The signals $ReqWrite$ and *ReqRead* are sent to the server part for the Read/Write request by the subject. Moreover, signals *Invalid* and *verified* sent out by AccessDeterm represent if an access request is approved or denied.

AccessDeterm's statechart diagram is illustrated as Fig. 5. The upper part of the diagram is the mediation of the `WriteOnly` permission, and the lower part is for the `Read-Only` permission. The decision rules are set by three state transition rules in DBLP, and described as guard conditions in the diagram.

The statechart diagram of the DBLP state transition is as Fig. 6. It is the implementation of Fig.3. The SelectObj sub-statechart diagram aims to generate an object's L_min_o, L_max_o and $level_c$ randomly.

4.3 Experiment and Result Analysis

We use the tool Hugo/RT in [20, 22] as the model translator and SPIN as the model checker, which is based on the linear temporal logic (LTL). Hugo/RT supports SPIN as a back-end to verify UML models. It also provides an intermediate format named UTE, which allows users to modify the result it generates, for its translation algorithm cannot deal with the LTL formula of SPIN well. So we rewrite the security properties to be checked in that form.

We ran the experiments on a Q9400 Core 2 Quad PC with 4GB memory which runs Fedora 14. The version of SPIN is v5.1.3. Considering the conformance of the UML specification, we use the UML v1.4.

Due to Hugo/RT does not support part of UML specifications well, we first generate the security policy model's UTE file by Hugo/RT and add the specification of confidentiality property manually to the UTE file as follows.

```
assertion {
    G (stat.read_level>stat.
        write_level);
}
```

in which the "G" is a LTL operator with the mean of "all the future state". Then, we convert the assert-added UTE file to PROMELA file using Hugo/RT again. Last, the PROMELA file was input to SPIN to check the violation of the goal of confidentiality. The output of SPIN is as follows:

```
warning: never claim + accept labels requires -a
flag fully verify hint: this search is more effi
cient if pan.c is compiled -DSAFETY pan: claim vi
olated! (at depth 492) pan: wrote dblp.spin.trail

(Spin Version 5.1.3 -- 11 May 2011) Warning:
Search not completed

Full statespace search for:
  never claim          +
  assertion violations + (if within scope of claim)
  acceptance   cycles  - (not selected)
  invalid end states   - (disabled by never claim)

State-vector 128 byte, depth reached 551, errors:
  1390 states, stored
  211 states, matched
  601 transitions (= stored+matched)
  590 atomic steps
hash conflicts:        0 (resolved)

  2.501 memory usage (Mbyte)


pan: elapsed time 0.01 seconds
```

We can conclude from the result that there exists a path, which violates the Definition 5, in the instance of DBLP

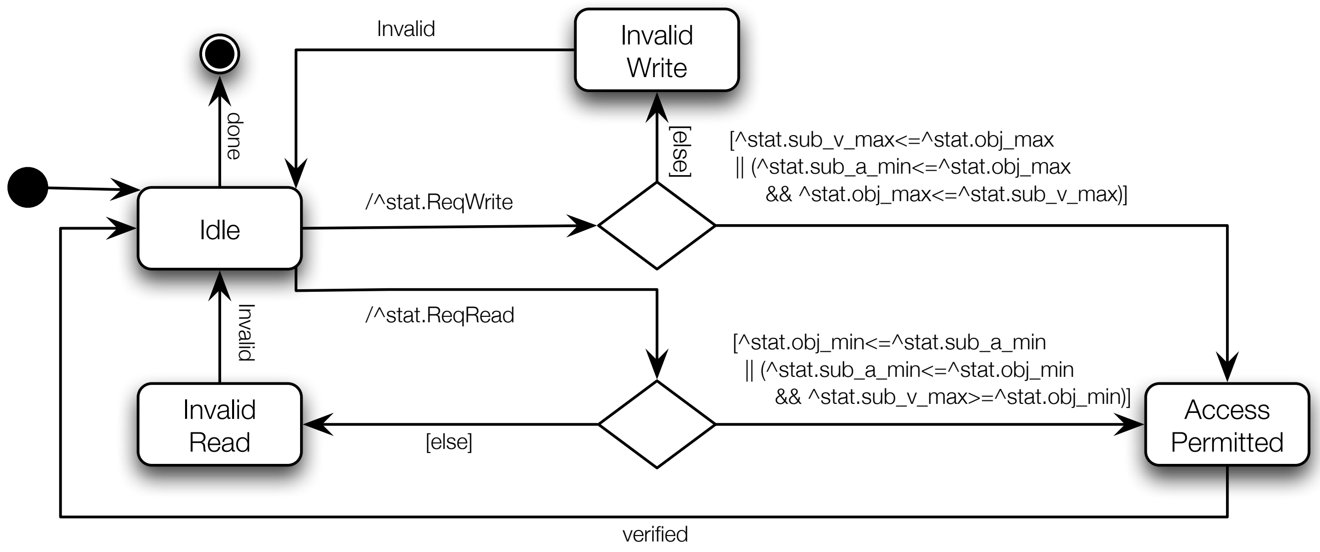
Invalid
Invalid Write
done
[else]
[^stat.sub_v_max<=^stat.obj_max || (^stat.sub_a_min<=^stat.obj_max && ^stat.obj_max<=^stat.sub_v_max)]
Idle
/^stat.ReqWrite
/^stat.ReqRead
Invalid
[^stat.obj_min<=^stat.sub_a_min || (^stat.sub_a_min<=^stat.obj_min && ^stat.sub_v_max>=^stat.obj_min)]
Invalid Read
[else]
Access Permitted
verified

Figure 5: UML statechart of the Mediation Part

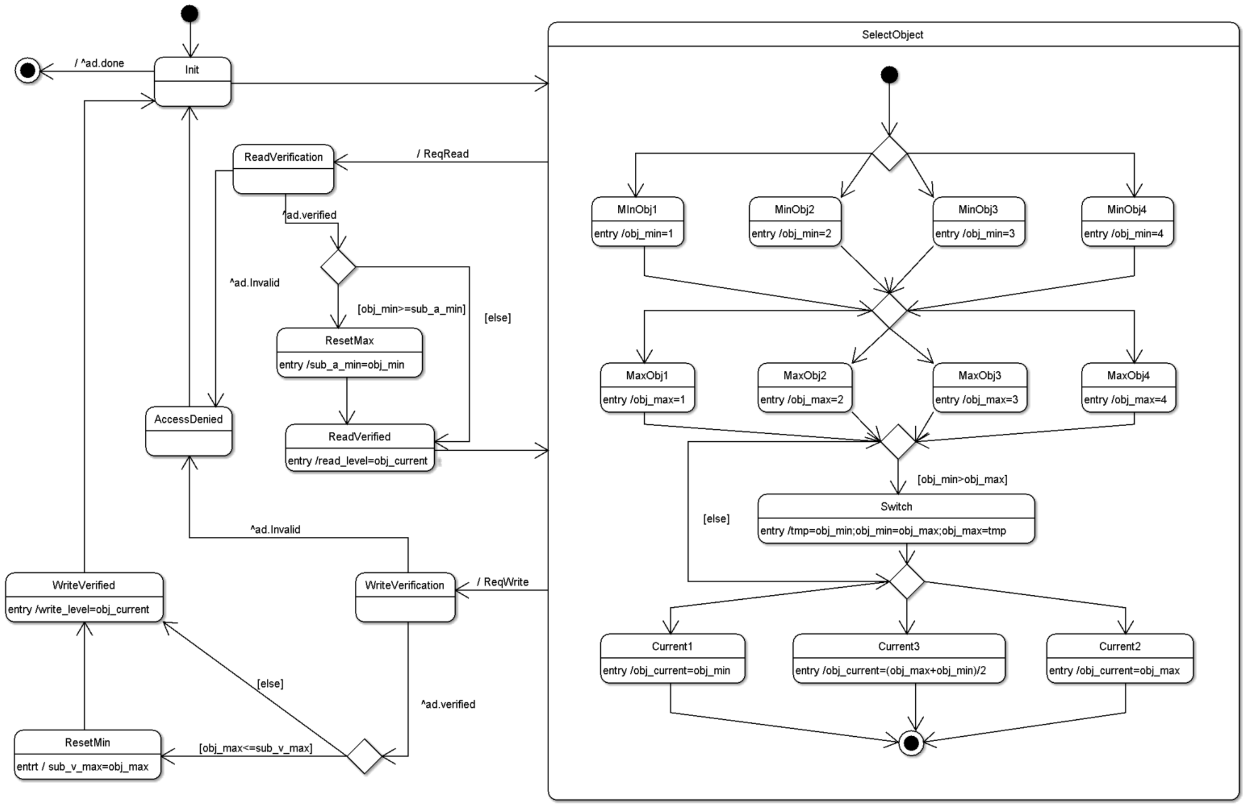

Figure 6: UML statechart of the System State Transition

model described in previous section. According to the trail file produced by the model checker SPIN, we can specify that path as follows:

- First, Subject S accesses the object $\{L\_min_o = 2, L\_max_o = 4, level_c(o) = 3\}$ with the `ReadOnly` permission, and its sensitivity level becomes$\{a_min_s = 2, v_max_s = 4\}$;
- Then, Subject S accesses the object $\{L\_min_o = 2, L\_max_o = 2, level_c(o) = 2\}$ with the `WriteOnly` permission. Since $a\_min_s(s) \leq L\_max_o(o) \leq v\_max_s(s)$, its sensitivity level turns out to be $\{a\_min_s = 2, v\_max_s = 4\}$ according to DBLP rules.

Note that, in the whole process, the non-trusted subject in the security policy model reads information from the object with the sensitivity level 3 and then writes to the object with sensitivity level 2. This sequence of actions causes the information flow from higher level to the lower level finally, which violates the confidentiality principle in the Definition 5. This result conforms to the conclusion in [23] and could demonstrate the effectiveness of our approach.

5. RELATED WORK

The idea of applying model checking on security policy models is not new. Hanen et al. [8] present an approach to verify the conformance between access control policy and its implementation by SPIN. They consider policies as LTL formulae and the implementation of policies as system behaviors. All their works are done by hand, and its efficiency is low. The PROMELA language of SPIN is also used by Kikuchi [9] to model system states and policies, in order to verify the consistence between them. Nevertheless, the "policy" mentioned in his approach is not security policy but more similar to a kind of system resource configuration. Moreover, his work of modeling is also done manually.

The translation from the UML state machine model to a model checker's input has been researched in [20] and proven to be correct and complete. Latella et al. [24] present a translation from a UML state machine to PROMELA for the first time. They also give the translation algorithm and demonstrate the correctness of their algorithm. Balser et al. [20] adopt Latella's algorithm and develop Hugo/RT to support different model checkers' input language.

Koch [2] proposes RBAC's policy specifications in UML notations and uses graph transformation to verify an example of RBAC policy. Park et al. [10] specify an example of RBAC policy by UML class diagrams and sequence diagrams and use Alloy and USE [11] tool to verify the RBAC instance half-formally. Ahn proposed AMF framework, which use class diagrams and object diagrams [12, 13] to model RBAC instance and OCL to represent security policies. However class diagrams and object diagrams are both static diagrams, so their UML model can only represent security model's static properties. If the violation occurs in the dynamic behavior sequence, their work seems not so capable. Indrakshi Ray's successive work [14, 15, 16] also realized the limitation of using UML static diagrams only, and they took sequence diagrams to depict the state transitions of security models. Their recent works in [15, 16] introduced the concept of Snapshot Model, which includes operations to trigger state transition and the before/after state of this operation. The snapshot can be generated from UML class diagrams automatically by tools. These generated *scenarios* would be labeled as "legal" or "illegal" by verifiers manually, then whether the scenarios are accepted by the design model or not is checked by tools. Ahn and Ray's works are both based on the Alloy tool also.

For comparison, our approach is based on model checker. Besides, we use state-machine diagrams to simulate the state transitions, and it does not have a procedure of labeling by hand. Moreover, most work above focus on validating the wrong configurations of a correct security model, and few of them has considered detecting flaws in security models themselves.

6. CONCLUSION AND FUTURE WORK

We propose a new verification method to check the correctness of a security policy model based on the study of predecessors' work in this paper. This approach employs a client/server mode to specify the security model from the view of access mediation. As an example for its usage, we present the verification of the DBLP policy model against the confidentiality property and demonstrate its violation of this principle by our approach. Our aim is to find an easy way for the operating system developers to understand the security policy model and verify the model without spending too much time on learning formal methods. However, we have not applied this method to a real system yet.

The approach has been proven to be capable of supporting analysis for a variety of checks, but only one property each time now. Since different properties have various appearances of assertion formulae, we have to add them into the intermediate output of the translator manually. For DBLP, the state transition model described in section 4.2 can only verify the property of confidentiality. In practice, we can model properties to different sub-statecharts in one orthogonal composite state to verify them concurrently. In the future work, we aim to the extensibility of our approach by constructing a property library. Moreover, we also plan to translate the trail file generated by SPIN into a more readable format for non-specialist readers.

7. ACKNOWLEDGMENTS

This work is supported by the National Science Foundation of China under Grants No. 60970028 and No. 61100227.

8. REFERENCES

[1] J. E. Tidswell and T. Jaeger. An access control model for simplifying constraint expression. In *7th ACM conference on Computer and Communications Security*, 2000.

[2] M. Koch and F. Parisi-Presicce. Visual specifications of policies and their verification. In *Workshop on Fundamental Approaches to Software Engineering.* LNCS 2621, 2003.

[3] Dury. Arnaud, Boroday. Sergiy, Petrenko. Alexandre, and Lotz. Volkmar. Formal verification of business workflows and role based access control systems. In *The International Conference on Emerging Security Information, Systems, and Technologies (SECUREWARE 2007)*, pages 201–210, Oct 2007.

[4] M. Drouineaud, A. Luder, and K. Sohr. A role based access control model for agent based control systems.

In *Proceedings of the IEEE International Conference on Industrial Informatics*, 2003.

[5] Kathi Fisler, Shriram Krishnamurthi, Leo Meyerovich, and Michael Tschantz. Verification and change impact analysis of access-control policies. In *International Conference on Software Engineering (ICSE)*, 2005.

[6] Object Management Group. *Unified Modeling Language Specification*. OMG, 2001.

[7] M. Koch, L. Macini, and F. Parisi-Presicce. Foundations for a graph-based approach to the specification of access control policies. In *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS 2001)*. Lect. Notes in Comp. Sci. Springer, March 2001.

[8] Hansen F. and Oleshchuk V. A. Conformance checking of rbac policy and its implementation. In *First Information Security Practice and Experience Conference (ISPEC 2005)*, pages 144–155, 2005.

[9] Kikuchi Shinji, Tsuchiya Satoshi, Adachi Motomitsu, and Katsuyama Tsuneo. Policy verification and validation framework based on model checking approach. In *Fourth International Conference on Autonomic Computing*, pages 1–1, 2007.

[10] Sachoun Park and Gihwon Kwon. Verification of uml-based security policy model. In *ICCSA*, pages 973–982, 2005.

[11] Richters M and Gogola M. Validating uml models and ocl constraints. In *Proceedings of the 3rd International Conference on the Unified Modeling Language (UML 2000)*, pages 265–277, York, UK, 2000.

[12] Ahn Gail-Joon and Hu Hongxin. Towards realizing a formal rbac modelin real systems. In *Proceedings of the 12th ACM Symposium on Access Control Models and Technologies(SACMAT)*, pages 215–224, New York, USA, 2007.

[13] Hu Hongxin and Ahn Gail-Joon. Enabling verification and conformance testing for access control model. In *Proceedings of the 13th ACM Symposium on Access Control Models and Technologies(SACMAT)*, pages 195–204, EstesPark, Colorado, USA, 2008.

[14] Yu Lijun, France Robert, and Ray Indrakshi. Scenario-based static analysis of uml behavioral properties. In *Proceedings of the ACM/IEEE 11th International Conference on Model Driven Engineering Languages and System*, pages 234–248, Toulouse, France, 2008.

[15] L. Yu, R. France, I. Ray, and S. Ghosh. A rigorous approach to uncovering security policy violations in uml designs. In *Engineering of Complex Computer Systems, 2009 14th IEEE International Conference on*, pages 126 –135, june 2009.

[16] Wuliang Sun, Robert France, and Indrakshi Ray. Rigorous analysis of uml access control policy models. In *IEEE International Symposium on Policies for Distributed Systems and Networks*, 2011.

[17] Jackson D. Alloy : A lightweight object modeling notation. *ACM Transactions on Software Engineering and Methodology*, 11(2):256–290, 2002.

[18] Ji Qing-Guang and He Ye-Ping Qing Si-Han. An improved dynamically modified confidentiality policies model. *Journal of Software*, 15(10):1547–1557, 2004.

[19] G. J. Holzmann. The spin model checker. *IEEE Trans. Softw. Eng*, 23:279–295, 1997.

[20] Michael Balser, Simon Bäumler, Alexander Knapp, Wolfgang Reif, and Andreas Thums. Interactive verification of uml state machines. In *Proc. 6th Int. Conf. Formal Engineering Methods (ICFEM'04)*, pages 434–448, 2004.

[21] Bell D E and La Padula L J. Secure computer system: A retrospective. In *Proceedings of the 1983 IEEE Symposium on Security and Privacy*, pages 161–162, 1983.

[22] Knapp Alexander and Merz Stephan. Model checking and code generation for uml state machines and collaborations. In *Proceedings of the 5th Workshop on Tools for System Design and Verification*, pages 59–64, 2002.

[23] He Jian-Bo. *Study on the key issues of security policy models in secure operating system*. PhD thesis, Institute of Software, Chinese Academy of Science, Beijing, 2007.

[24] Latella D, Majzik I, and Massink M. Automatic verification of a behavioural subset of uml statechart diagram using spin model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.