Property Verification for Access Control Models via Model Checking[1]

Vincent C. Hu[1], D. Richard Kuhn[1], Tao Xie[2]
[1]*Institute of Standards and Technology,* [2]*Noth Carolina State University*
vhu@nist.gov, kuhn@nist.gov, xie@csc.ncsu.edu

Abstract

To formally and precisely capture the security properties that access control should adhere to, access control models are usually written, bridging the rather wide gap in abstraction between policies and mechanisms. In this paper, we propose a new general approach for property verification for access control models via model checking. The approach defines a standardized structure for access control models, providing for both property verification and automated generation of test cases. The approach expresses access control models in the specification language of a model checker and expresses generic access control properties in the property language. Then the approach exploits the model checker to verify these properties for the access control models and generate test cases for the system implementations of the models.

1. Introduction

Access control systems are among the most critical of network security components. It is common that a system's privacy and security are compromised due to the misconfiguration of access control policies instead of the failure of cryptographic primitives or protocols. This problem becomes increasingly severe as software systems become more and more complex, and are deployed to manage a large amount of sensitive information and resources that are organized into sophisticated structures. Identifying discrepancies between policy specifications and their intended function is crucial because correct implementation and enforcement of policies by applications is based on the premise that the policy specifications are correct. As a result, policy specifications must undergo rigorous verification and validation to ensure that the policy specifications truly encapsulate the desires of the policy authors.

To formally and precisely capture the security properties that access control should adhere to, access control models are usually written, bridging the rather wide gap in abstraction between policies and mechanisms: users see access control models as an unambiguous and precise expression of requirements; vendors and system developers see access control models as design and implementation requirements.

In this paper, we propose a new general approach to property verification for access control models via model checking. The approach first expresses access control models in the specification language of a model checker and expresses generic access control properties in temporal logic formulas. Then the approach exploits the model checker to verify these properties for the access control models and generate test cases to check the conformance of the models and their implementations.

2. Approach

Our approach expresses access control models in the specification language of a model checker (Section 2.1), expresses generic access control properties in temporal logic formula, and verifies these properties with the model checker (Section 2.2). *Test cases*, consisting of *input data* and *expected results*, are generated following verification (Section 2.3). One goal of the techniques in our approach is to reduce overall software assurance costs by integrating verification with test generation.

2.1. Model Specification

Our approach specifies access control models with Finite State Machines (FSMs), which range from completely synchronous to completely asynchronous ones, and from detailed to abstract ones. The only data types in the specification are finite– booleans, scalars, fixed arrays, and static data types that can be constructed [1]. The specified model in an FSM describes the transition states of the FSM. In general, any expression in the propositional calculus can be used to define the transition relation of states; however, the flexibility of the expression is accompanied by the risk of a logical contradiction, which makes specifications vacuously true or makes the system unimplementable. Fundamentally, there are three basic types of FSM expressions for specifying access control models in terms of the sequence of the state transitions:

Synchronous. An access control model is expressed by defining the value of access control constraints in

[1] This work is supported in part by NSF grant CNS-0716579.

the next state (i.e., after each transition), given the value of constraints in the current states (i.e., before transition).

Asynchronous. An access control model is expressed by a collection of concurrency states. This type of model is for access control systems whose authorization decisions are triggered from more than one clock region such as mutual exclusion, communication protocols, and asynchronous circuits.

Direct specification. An access control model is specified directly in terms of propositional formulas. The set of possible initial states is specified as a formula in the current state variables. A state is *initial* if it satisfies the formula. The transition relation is directly specified as a propositional formula in terms of the current and next values of the state variable. Any current state/next state pair is in the transition relation if and only if it satisfies the formula.

2.2. Property Specification and Verification

From the separation of duty and safety point of view, the state transitions of access control properties can be categorized by three types of constraints, namely static, dynamic, and historical constraints. We next illustrate how these constraints (together with their models) can be specified and verified by our approach.

2.2.1. Static Constraints. Static constraints regulate the access permission by static system states or conditions such as rules, attributes, and system environments (time and locations for access). Popular access control models with these types of properties include Role-Based Access Control (RBAC) [2] and Multi-level access control [3]. These types of models can be specified by **asynchronous** or **direct specification** expressions of an FSM. The transition relation of authorization states is directly specified as a propositional formula in terms of the current and next values of the state variables. Any current state/next state pair is in the transition relation if and only if it satisfies the formula, as demonstrated in the following **direct specification** of an FSM:

```
{
  VARIABLES
     access_state : boolean;  /* 1 as grant, 0 as deny*/
     ……..
  INITIAL
    access_state  = 0;
  TRANS /* transit to next access state */
    next (access_state) =
      ( (constraint_1 & constraint_2 & …… constraint_n) |
      (constraint_a & constraint_b & …… constraint_m) ……..)
}
```

where the system state of access authorization is initialized as the ***deny*** state and moved to the ***grant*** state for any access request that complies with the constraints of the rule corresponding with each constraint predicate (i.e., *constraint_i*….& *constraint_n*), and stay in the ***deny*** state otherwise. The properties of the static constraints can be verified by verifying the properties expressed in the following temporal logic formula:

AG (constraint_1 & constraint_2 & …. constraint_n) → AX (access_state = 1)
AG (constraint_a & constraint_b & …. constraint_m) → AX (access_state = 1) ……
AG ! ((constraint_1 & ….constraint_n) | (constraint_a & …. constraint_m) |...) → AX (access_state = 0)

which simply means that all access requests that comply with specified constraints for the rules should be granted for access, and all non-complied ones should be denied.

2.2.2. Dynamic Constraints. Dynamic constraints regulate the access permission by dynamic system states or conditions such as specified events or system counters. An access control model with these types of properties specifies that accesses are permitted only by a certain subject to a certain object with certain limitations (e.g., object *x* can be accessed only no more than *i* times simultaneously by user group *y*). For example, if a user's role is a *cashier*, he or she cannot be an *accountant* at the same time when handling a customer's checks. This type of model can be specified with **asynchronous** or **direct specification** expressions of an FSM, which use a variable semaphore to express the dynamic properties of the authorization decision process. Another example of dynamic constraint states is enforcing a limited number of concurrent accesses to an object. The authorization process for a user thus has four states: ***idle, entering, critical***, and ***exiting.*** A user is normally in the ***idle*** state. The user is moved to the ***entering*** state when the user wants to access the critical object. If the limited number of access times is not reached, the user is moved to the ***critical*** state, and the number of the current access is increased by 1. When the user finishes accessing the critical object, the user is moved to the ***exiting*** state, and the number of the current access is decreased by 1. Then the user is moved from the ***exiting*** state to the ***idle*** state. The authorization process can be modeled as the following **asynchronous** FSM specification:

```
{
  VARIABLES
     count, access_limit : INTEGER;
     request_1 : process_request (access_limit);
     request_2 : process_request (access_limit);
     …….
     request_n: process_request (access_limit);
 /*max number of user requests allowed by the system*/
     access_limit := k;
     count := 0;
```

```
process_request  (access_limit) {
    VARIABLES
        permission : {start, grant, deny};
        state : {idle, entering, critical, exiting};
    INITIAL_STATE (permission) = start;
    INITIAL_STATE (state) := idle;
    NEXT_STATE (state) := CASE {
        state == idle : {idle, entering};
        state == entering & ! (count > access_limit) :
                                                critical;
        state == critical : {critical, exiting};
        state == exiting : idle;
        OTHERWISE: state; };
    NEXT_STATE (access_limit) := CASE {
        state == entering : access_limit + 1;
        state == exiting : access_limit -1;
        OTHERWISE: DO_NOTHING;      };
    NEXT_STATE  (permission) = CASE {
        state == entering: grant;
        OTHERWISE: deny; }
  }
}
```

The state variables of the preceding example are used as the asynchronous states for the concurrent access of the limited number of access request. The specification of the dynamic constraints is verified by verifying the following properties expressed in temporal logic formula:

```
AG ! (state == entering | state == idle | state == critical | state ==
exiting)) → G (access = grant)
AG (state == idle | state == critical | state == exiting) →
G (access = deny)
```

where temporal logic formula $AG\,(p \rightarrow G\ q)$ indicates that "if condition p is true at time t, condition q is true at all times later than t.

2.2.3. Historical Constraints. Historical constraints regulate the access permission by historical access states or recorded and predefined series of events. The representative access control policies for this type of access control models are N-person [4], Chinese Wall [5], and Workflow [6] access control policies. This type of models can be best described by **synchronous** or **direct specification** expressions of an FSM. For example, the following **synchronous** FSM specification specifies a Chinese Wall access control model where there are two Conflict of Interested group of objects:

```
{
    VARIABLES
        access {grant, deny};
        act {rd, wrt};
        object {none, COI1, COI2};
        state {1, 2, 3}
    INITIAL_STATE(state) := 1;
    INITIAL_STATE(object) := none;
    NEXT_STATE(state) := CASE {
        state == 1 & act == rd & object == COI1: 2;
        state == 1 & act == rd & object == COI2: 3;
        state == 2 & act == rd & object == COI1: 2;
        state == 2 & act == rd & object == COI2: 2;
        state == 3 & act == rd & object == COI1: 3;
        state == 3 & act == rd & object == COI1: 3;
        OTHERWISE: 1; };
   NEXT_STATE(access) := CASE {
        state == 2 & act == rd & object == COI1: grant;
        state == 3 & act == rd & object == COI2: grant;
        OTHERWISE: deny; };
    NEXT_STATE (act) = act;
    NEXT_STATE (object) = object;
}
```

The properties of the dynamic constraints can be verified by verifying the following temporal logic formula:

```
AG ((state == 2 & act == rd & object == COI1) | (state == 3 & act ==
rd & object == COI2)) → AX (access = grant)
AG ! ((state == 2 & act == rd & object == COI1) | (state == 3 & act ==
rd & object == COI2)) → AX (access = deny)
```

2.3. Test Generation

In addition to supporting property verification, the model checking technique was adopted because it fits well with a variety of test generation techniques, such as fault-based mutation testing [7] and combinatorial testing [8]. Mutation testing allows us to test for the presence of hypothesized faults, or faults that they subsume, and combinatorial testing makes it possible to rule out complex interactions that may lead to failures. As testing must always be conducted once a policy is implemented to assure correct implementation, automated generation of test cases can reduce total costs, thus making formal specification easier to integrate into the development process. Model checking is ideal for this integration because it can solve the oracle problem for testing (determining expected results for a particular set of test input data), in addition to formal verification of properties. A case study of this technique for software is given in [9].

Even with highly automated tools such as model checkers, real-world development budgets rarely allow the development and exploration of formal models, because the cost must be balanced against the cost of releasing code with errors that would not be caught in testing. But testing typically consumes 50% or more of a development budget. Generating test cases from formal specifications makes it cost-effective to allocate a portion of the testing budget to produce a formal specification, which can then be used to confirm desired properties and generate test cases.

To produce test cases that guarantee combinatorial coverage to an interaction level t, we produce a t-way covering array for input parameters used in the policy. Informally, a covering array can be viewed as a table of input data where each column is an input parameter and values in each column are parameter values, so that each row represents a test. All possible t-way combinations of parameter values are guaranteed to be covered at least once. If $t = 2$, this procedure results in the familiar "pairwise" testing, but using new algorithms [10], we are able to produce covering arrays up to strength $t = 6$.

Two *specification claims* are generated for each covering array row, one for result *grant*, and one for result *deny*. Values v_{ij} are taken from row $_i$, column $_j$ of the covering array, for all rows.

$AG\ (p_1 = v_{11}\ \&\ \ldots\ \&\ p_n = v_{n1}) \rightarrow AX\ !(access\_state = grant)\ \ldots\ldots$

$AG\ (p_1 = v_{12}\ \&\ \ldots\ \&\ pn = v_{n2}) \rightarrow AX\ !(access\_state = grant)\ldots\ldots$

$AG\ (p_1 = v_{11}\ \&\ \ldots\ \&\ p_n = v_{n1}) \rightarrow AX\ !(access\_state = deny)\ldots\ldots$

For a covering array with n rows, a total of $2n$ specification claims will thus be produced, one *grant* and one *deny* for each row of the covering array. In the claims, possible results *grant* or *deny* are negated. For each claim, if this set of values cannot in fact lead to the particular result, the model checker indicates that this is true. If the claim is false, the model checker indicates so and provides a *counterexample* with a trace of parameter input values and states that will prove it to be false. The model checker thus filters the claims that we have produced so that a total of n test inputs is generated. In effect, each one is a test case, i.e., a set of input parameter values and expected result. It is then simple to map these values into test cases in the syntax needed for the system under test. When interaction testing is done today, t is nearly always 2 (i.e., pairwise testing), because higher strength interactions require exponentially more test cases. Thus higher strength interaction testing requires fully automated generation of test input data and expected results, which is made possible through model checking.

This technique makes it possible to produce two complementary types of test cases. In addition to combinatorial test cases, fault-based testing can be automated. By inserting particular faults in the specification, then generating counterexamples using the model checker, we can produce test cases that will detect these faults or faults that are subsumed by them.

3. Related Work

There exist several verification techniques for applying model checking on access control *policies* but few *general* verification techniques for applying model checking on access control *models* and generating test cases as our proposed approach. Zhang et al. [11] models rule-based policies in their RW language based on propositional logic and conduct model checking on the policies. Kikuchi et al. [12] model both an application system and its policy with the input language of an explicit state model checker and conduct model checking. Schaad et al. [13] use NuSMV to verify RBAC policies for workflows against properties of separation of duties. Different from these existing approaches, our proposed approach is targeted at access control models and their generic properties, and is more general and applicable in a larger scope of models and properties. In addition, our approach provides test generation besides property verification.

4. Conclusion

To verify properties for access control models, we propose a new general approach that expresses access control models in the specification language of a model checker and generic access control properties in its property language as temporal logic formula. Then the approach exploits the verification process of the model checker to verify the specified models against the specified properties. Our approach is able to support the verification of three common types of generic access control properties: static, dynamic, and historical constraints. In addition, the approach also supports automated generation of test cases to check the conformance of the models and their implementations.

References

[1] NuSMV: a new symbolic model checker. http://nusmv.irst.itc.it/

[2] D. Ferraiolo and R. Kuhn. Role based access control. In *Proc. 15th NIST-NCSC National Computer Security Conference*, pp. 554–563, 1992.

[3] D. E. Bell and L. J. LaPadula. Secure computer systems: Mathematical foundations, 1973. MITRE Corporation.

[4] National Computer Security Center. Integrity in Automated information System. Technical Report 79-91, Library No. S237,254, Sept. 1991.

[5] D. F. C. Brewer and M. J. Nash. The Chinese wall security policy. In *Proc. IEEE Symposium on Security and Privacy*, pp. 206–214, 1989.

[6] Workflow Management Coalition. Workflow Management Coalition Terminology & Glossary. http://www.wfmc.org/ Documentation number WFMC-TC-1011, February 1999.

[7] P. Ammann and P.E. Black. Abstracting Formal Specifications to Generate Software Tests via Model Checking. In *Proc. Digital Avionics Systems Conference*, pp. 10.A.6-1 - 10.A.6-10, 1999.

[8] D.R. Kuhn, D.R. Wallace, and A.J. Gallo, Jr. Software Fault Interactions and Implications for Software Testing. IEEE Trans. on Software Engineering, Vol. 30, No. 6, June 2004.

[9] D. R. Kuhn, V. Okun. Pseudo-exhaustive Testing For Software, In *Proc. 30th NASA/IEEE Software Engineering Workshop*, April 25-27, 2006.

[10] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, and J. Lawrence. IPOG/IPOG-D: Efficient Test Generation for Multi-Way Combinatorial Testing, Software Testing, Verification, and Reliability. To appear.

[11] N. Zhang, M. D. Ryan, and D. Guelev. Evaluating Access Control Policies Through Model Checking. In *Proc. Information Security Conference*, pp. 446-460, 2005.

[12] S. Kikuchi, S. Tsuchiya, M. Adachi, and T. Katsuyama. Policy Verification and Validation Framework Based on Model Checking Approach. In *Proc. International Conference on Autonomic Computing*, pp. 1-9, 2007.

[13] A. Schaad, V. Lotz, and K. Sohr. A model-checking approach to analysing organisational controls in a loan origination process. In *Proc ACM Symposium on Access Control Models and Technologies*, pp. 139-149, 2006.