# General Methods for Access Control Policy Verification

Vincent C. Hu, D. Richard Kuhn
National Institute of Standards and Technology
Gaithersburg, MD, USA
vhu, kuhn@nist.gov

***Abstract*— Access control systems are among the most critical of computer security components. Faulty policies, misconfigurations, or flaws in software implementations can result in serious vulnerabilities. To formally and precisely capture the security properties that access control should adhere to, access control models are usually written, bridging the gap in abstraction between policies and mechanisms. 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 represented by models must undergo rigorous verification and validation through systematic verification and testing to ensure that the policy specifications truly encapsulate the desires of the policy authors. Verifying the conformance of access control policies and models is a non-trivial and critical task, and one important aspect of such verification is to formally check the inconsistency and incompleteness of the model and safety requirements of the policy, because an access control model and its implementation do not necessarily explicitly express the policy, which can also be implicitly embedded by mixing with direct access constraints or other access control models.**

***Keywords*— *Access Control, Authorization, Policy, Policy Verification, Policy Testing, Policy Tool, Model Checking.***

## I. INTRODUCTION

Access control (AC) systems control which users or processes have access to which resources in a system. They are among the most critical of computer security components. AC policies are specified to facilitate managing and maintaining AC systems, therefore faulty policies, misconfigurations, or flaws in software implementation can result in serious vulnerabilities. However, the correct implementations of AC policies by AC mechanisms are very challenging problems. It is common that a system's privacy and security are compromised due to the misconfiguration of AC 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. Therefore, identifying discrepancies between AC 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.

AC models are usually written to bridge the rather wide gap in abstraction between AC policies and mechanisms to formally and precisely capture the safety requirements that AC systems should adhere to. As a result, policy specifications represented by models must undergo rigorous verification and validation through systematic verification and testing to ensure that the policy specifications truly encapsulate the desires of the policy authors. Verifying the conformance of AC policies and models is a non-trivial and critical task. One important aspect of such verification is to formally check the inconsistency and incompleteness of the model and policy safety requirements, because an AC model and its implementation do not necessarily explicitly express the policy, which can also be implicitly embedded by mixing with direct access constraints or other AC models.

In this document, we discuss general approaches for the verification for AC models and the testing of model implementations by first defining standardized structures of AC models. We then demonstrate the expressions of AC models and safety requirements in formal specifications of model checkers for the use of black box and white box model verifications that verify the integrity, coverage, and confinement of the specified safety requirements against models. In addition, an efficient way of generating test cases for the implementation from a model is discussed.

This document is divided into seven sections. Section I states the purpose, of this document. Section II introduces the general concept of AC policy and model. Section III explains the elements of AC safety and faults. The focus of this document is presented in Section IV, which introduces main concepts for AC model verification and testing. Section V provides some AC system implementation considerations. Section VI present some major related works. Section VII is the conclusion to the document.

## II. GENERAL AC MODELS

An AC model is a formal presentation of an AC policy enforced by the mechanism and is useful for proving theoretical limitations of an AC system so that AC

mechanisms can be designed to adhere to the properties of the model. Users see an AC model as an unambiguous and precise expression of requirements. Vendors and system developers see AC models as design and implementation requirements. On one extreme, an AC model may be rigid in its implementation of a single policy. On the other extreme, an AC model will allow for the expression a wide variety of policies and policy classes. In general, all **nondiscretionary** AC polices can be modeled by static, dynamic and historical Finite State Machine (FSM) models from one of the following classes:

*a) Static model*

Static policies regulate the access permission by static system states or conditions such as rules, attributes, and system environments (times and locations for access). Popular AC policies with these types of properties include ABAC [1], MLS[2], and RBAC[2]. These types of policies can be specified by **asynchronous** or **direct specification** expressions of an FSM model. 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 Example 1:

```
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)
  ……..);
```

**Example 1 – static AC model**

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_1.*& *constraint_n*) in a rule, and stay in the ***deny*** state otherwise.

*b) Dynamic model*

Dynamic policies may include temporal constraints that regulate access permissions by dynamic system states or conditions such as specified events or system counters or N-person AC policy. An AC 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 policy can be specified with **asynchronous** or **direct specification** expressions of an FSM model, which uses 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; example 2:

```
VARIABLES
    count, access_limit : INTEGER;
    request_1 : process_request (count);
    request_2 : process_request (count);
    …….
    request_n: process_request (count);
    /*max number of user requests allowed by the system*/
    access_limit := k;  /*max number of concurrent access*/
    count := 0; act {rd, wrt}; object {obj};
    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 (count) := CASE {
           state == entering : count + 1;
           state == exiting : count -1;
           OTHERWISE: DO_NOTHING };
       NEXT_STATE (permission) := CASE {
          (state == entering) & (act == rd) & (object == obj):
grant;
           OTHERWISE: deny;
           }
    }
```

**Example 2 – dynamic AC model**

*c) Historical model*

Historical policies regulate access permissions by historical access states or recorded and predefined series of events. Representative AC policies for this type of AC policies including Chinese Wall [2] and Workflow AC [2] policies. This policy class can be best described by **synchronous** or **direct specification** expressions of an FSM model. For example, the following Example 3 synchronous FSM specification specifies a Chinese Wall AC policy where there are two Conflict of Interest groups *COI1*, *COI2* of objects:

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

```
    u_state == 2 & act == rd & o_state == COI2: 2;
    u_state == 3 & act == rd & o_state == COI1: 3;
    u_state == 3 & act == rd & o_state == COI2: 3;
    OTHERWISE: 1; };
NEXT_STATE(access) := CASE {
    u_state == 2 & act == rd & o_state == COI1: grant;
    u_state == 3 & act == rd & o_state == COI2: grant;
  OTHERWISE: deny; };
  NEXT_STATE (act) := act;
  NEXT_STATE (o_state) := object;
```

**Example 3 – historical AC model**

Note that in practice, the same AC policies may be expressed by multiple different AC models or expressed by a single model in addition to extra constraint rules outside of the model.

## III. AC SAFETY AND FAULTS

Safety is the fundamental property of an AC system, which ensure that the AC system will not result in the leakage/blockage of permissions to an unauthorized/authorized principal. Thus, an AC system is safe if no privilege can be escalated to unauthorized or unintended principals, but the correct privileges are always accessible to authorized principals. Safety is specified through the use of restricted AC models that can be proven in general for that model describing the safety requirements of any configuration [3].

Among all the safety features, **Separation of Duties** (SoD) [2] are more dynamic than others. SoD refers to the principle that no user should be given enough privileges to misuse the system on their own. For example, the person authorizing paychecks should not also be the one who can prepare them. SoDs can be enforced either statically (by defining conflicting roles, i.e., roles which cannot be assigned to the same user) or dynamically (by enforcing the control at access time). AC faults compromise the safety, at semantic level, AC faults are usually caused by erroneous or inefficient representation of AC properties or permission algorithms. At a syntactic level, AC faults are simply caused by implementation errors in AC mechanism such as coding errors, or misconfigurations of AC systems. In general, AC faults can be categorized into the following classes.

***Privilege leakage***

Privilege (i.e. action and resource pair) leakage refers to situations in which subject is able to access resources that are prohibited by the safety requirements. Such leakage may cause either the privilege escalation from one resource domain or class to prohibited ones such as leakage from lower to higher ranks of MLS policy, or privilege leak such as from one role to other prohibited ones of an RBAC policy. Privilege leakage can be caused by mistaken privilege assignment directly or careless privilege inheritance indirectly.

***Privilege blocking***

Opposite to privilege leaking, a privilege blocking fault blocks a legitimate access to rightful resources. Privilege blocking can also occur when the properties of AC policy cannot render a *grant* or *deny* decision, or there is no available logic in the AC policy algorithm for evaluating the access request. Privilege blocking can also be a result of the deadlock of access rules specification where: a rule has a dependency on other rule(s), which eventually depend back on the rule itself such that a subject's request will never reach a decision because of the cyclic referencing.

***Cyclic inheritance***

Cyclic inheritance fault refers to the problem of privileges inheritance from other subjects(groups), which also in a chain of inheritance relation inherit back to the subject(group)'s privilege. For example, subject *x* inherits privilege from subject *y*, which inherit privilege from subject *z*, which inherits privilege from subject *x*. Cyclic inheritance leads to undecidable or infinite access evaluation process.

***Privilege conflict***

Unlike regular programming logic that a later value assignment of a variable overwrites the previous assigned value of the same variable, the rules of an AC policy normally have no precedence consideration in permission evaluation. In other words, AC rules will not be overwritten by other rules unless specifically allowed to. Thus, privilege conflicts appear when the specifications of two or more access rules result in the conflicting decisions of permitting subjects access requests by either direct or indirect (inherit) access assignments. In addition, when multiple policies are evoked for permission, conflicting decisions between policies may occur.

***Multi-policies considerations***

In an enterprise environment, it may be required to have AC policies specified independently by different collaborative or networked systems in the enterprise. Thus, an inter-system access request may be evaluated by more than one policy that the requesting subject is governed under. Thus, AC policy autonomy should also be preserved for secure inter-system access. Maintaining the autonomy of all collaborative system is a key requirement of the policy for inter-operation. The principle of autonomy states that if an access is permitted by an individual system, it must also be permitted under secure inter-system access. The principle of security states that if an access is denied by an individual system, it must also be denied under secure inter-system access. In a collaborative system, violations of secure inter-system access can be caused by adding inter-system privilege inheritance relations, for example, Fig. 1 shows that privilege *k* inherits privilege *j* through legal inter-system privilege inheritance (because both has the same privilege level *j*), which is granted in network *x* but denied in network *y*. These types of violations can be detected by checking for cyclic inheritance, privilege leakage and SoD violation. Thus, both security and autonomy can be characterized as safety requirements of a multi-policies AC system, which should be preserved during collaborations. A meta-policy is a policy that is usually applied for reconciling policy autonomy difference or to handle priorities of access decisions rendered from more than one policy. Thus, in addition to autonomy requirements, AC safety requirement may include priority model within the meta policy [4].

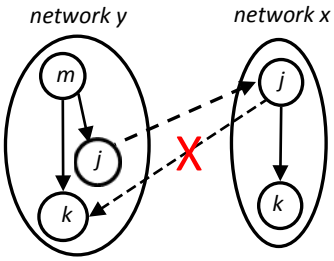


Fig.1 Privilege leaks through inter-system privilege inheritance.

## IV. Verification Approaches

The fundamental goal of AC policy and implementation verification is to detect conflicting or missing rules (i.e. policy statements) by verifying AC policy model and testing output of the policy. To achieve this, semantically and syntactically methods with Black-box and/or White-box testing techniques may be used. Although the general safety computation is proven undecidable [5] especially for discretionary AC policies, which are impossible to be described by static policy models, practical safety constraints such as confinements can be specified for discretionary AC policies. As a result, verifications can be performed upon the constraints.

In a nutshell, AC policy verification must test if the **safety requirements** of an AC policy are incorporated in the expressed model, which will be the blue print for implementing the AC system. The specification of safety requirements can be AC properties, business requirements, specifications of expected/unexpected system security features, or direct translations of policy features. Safety requirements can also include privilege inheritance, for example to verify a SoD property, safety requirement will specify that 1) subject *x* and *y* are mutually exclusive if neither one inherit the other's privilege directly or indirectly, 2) If subject *x* and *y* are mutually exclusive, then there is no other subject inherits privilege from both of them. Similar to SoD, dynamic SoD (DSoD) has the safety requirement: 3) If SoD holds, then DSoD is maintained. Thus, 1) and 2) must be guaranteed [4].

Note that an AC policy is not necessarily explicitly expressed by a single model; it can also be implicitly embedded by mixing with direct access constraints or other AC models. Thus, an AC policy may be expressed by combining multiple AC models (e.g. for policy combinations) or additional constraints outside of the model into one combined model. The principle of ensuring the conformance of a model to the policy is to formally detect inconsistency and incompleteness faults as described in Section III. In the former case, for example, an access request can be both accepted and denied, while in the latter case the request is neither accepted nor denied according to the model.

### *Model Verification*

The general approach for checking the correct specification of an AC model is to use black-box methods to verify the AC model against safety requirements. And since the confidence of the model's correctness depends on the quality of the safety requirements, a white-box property assessment method on entities in the model and safety requirements is required to assess the sufficiency of the safety, covering and confinement of the model [6].

In terms of AC attributes the formal definition of AC model can be illustrated by a deterministic finite state transducer of a model corresponding to a Finite State Machine (FSM) with a five-tuple $M = (\Sigma, ST, s_0, \delta, F)$, where $\Sigma$ is the input alphabet that represents the attributes associated with subjects, actions, objects, and environment conditions.
$ST$ is a finite, non-empty set of recorded AC system states and permissions, $s_0$ is the initial state, $\delta$ is the state-transition function, where $\delta : ST \times \Sigma \rightarrow ST$, $F$ is the set of final states include *Grant*, *Deny* as the output.

For **static** AC models as described in II a, the FSM $M_{static}$ does not require intern states to reach the permission state, thus $F = ST = \{Grant, Deny\}$, i.e., $M_{static}$ is just a straightforward FSM model without state transitions. For **dynamic** AC models as described in II b, the input alphabets of FSM $M_{dynamic}$ are $\Sigma_{dynamic} = \{gCond_1, \ldots, gCond_n\}$, where global condition $gCond_i$ is the threshold indicator of the access limitation, such as the number of persons that have to access at the same time in a N-Person control policy [2], or the maximum number of accesses allowed for a Limited_Number_of_Access policy. For historical AC models as described in II c, the input alphabets of the FSM $M_{historical}$ are $\Sigma_{historical} = \{\ldots.. sCond_i, aCond_i, oCond_i \ldots\}$, where subject condition $sCond_i$ action condition $aCond_i$ and object condition $oCond_i$ contribute to a historical event that is used as determining factors for the next permission decision. Note that it is possible for different types of AC models to combine into one model such that $M_{combine} = \{M_{static} \cup M_{dynamic} \cup M_{historical}\}$[2].

An AC safety requirement $p$ is expressed by the proposition $p$: $ST \times \Sigma^2 \rightarrow ST$ of FSM, which can be collectively translated in terms of logical formulae such that $p = (sCond_1 * \ldots * sCond_n * aCond_1 * \ldots * aCond_n * oCond_1 * \ldots * oCond_n * gCond_1 * \ldots * gCond_n) \rightarrow d$, where $p \in P$ and $d$ is the permission is a set of safety requirements, and * is a Boolean operator in terms of logical formulas of temporal logic such as computational tree logic (CTL) and linear-time temporal logic (LTL). The purpose of model checking is to verify the set $ST$ in $M$ in which $p$ is true according to an exhaustive state space search. In addition, by verifying the set of states in which the negation of $p$ is true, we can obtain the set of counterexamples to make the assertion that $p$ is true. The satisfaction of an AC model $M$ to the AC safety requirement $P$ by model checking is composed of two requirements:
(1) Safety, where $M$ satisfies $P$. That is, there is no violation of rules to the logic specified in $P$, and it is assured that $M$ will eventually be in a desired state after it takes actions in compliance with a user access request.
(2) Liveness, where $M$ will not have unexpected complexities. That is, there is neither a deadlock in which the system waits

forever for system events, nor a **livelock** in which the model repeatedly executes the same operations forever.

Thus, the AC rules define the system behaviors that function as the transition relation $\delta$ in $M$. Then when the AC safety requirement is represented by temporal logic formula $p$, we can represent the assertion that model $M$ satisfies $p$ by $M \models Ap \rightarrow AXd$, where temporal logic quantifier $A$ represents "always", and logic quantifier $X$ represents "is true next state". The purpose of safety and liveness verification using model checking is to determine whether these assertions are true, and to identify a state in which the assertions are not true as a counterexample for the assertions. Since the behavior of the AC mechanism can be represented by FSM $M$, and the safety requirements that $M$ must satisfy can be represented by temporal logic formulas, we can define the correctness more precisely as that the model can be led from every possible state that is reachable from initial states to the defined final state while complying with the safety requirements.

Even though checked by the black box testing as described above, the model is not fault proof because the temporal logic in the model might not be thorough in covering all possible values of all rules or all conditions in rules. For example, two states determined by opposite assignments of the same Boolean variable are embedded in different sub-state modules, where a third state is triggered only when the constraints of the two states are satisfied. As demonstrated in Fig. 2, the two rules will never agree due to the self-negation to the same constraint. In this case, the third state will never be satisfied, but proven correct without counterexamples through the black box checking.

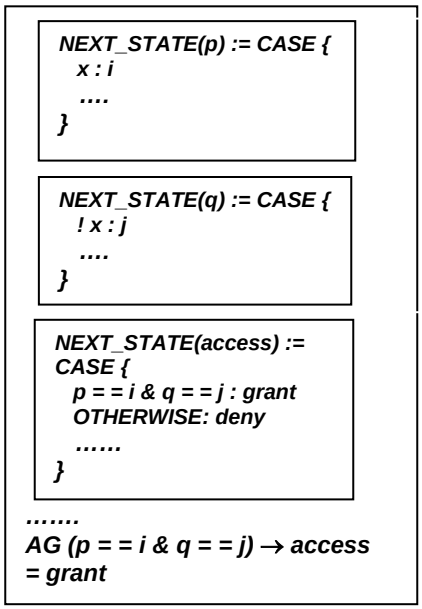


Fig.2 Example of never-achieved rules and the safety requirement in an AC model.

To detect this kind of semantic fault, the white box testing, based on code analysis should be applied such that the resulting mutated versions are used to detects faults of the model. Testing for mutations makes sure all paths of a part of a model code are covered by setting the related target variables to all possible values as input, and checking to see if there are different outcomes from the changes. If there is none, then either the code that had been mutated was never executed or the variable was unable to locate the faults. As shown in Fig. 2, If we mutate the first ***case*** module to change $x$ to $!x$, the resulting *access* state will be *grant* without being affected. (That works the same for the second case module). This fault demonstrates that there is a redundancy in the model, which does not violate the temporal logic of the model. Further investigation to check the model that relates to the variable should reveal that the ***(p = = i & q = = j) → access = = grant*** safety requirement will never happen. Note that this fault can be caught if one more safety requirement ***E! (p = = i & q = = j)*** (which means there exists some path that eventually in the future will satisfy ***!(p = = i & q = = j)*** in CTL model checking) is specified for the black box checking. Hence, it is not expected that all safety requirements are perfectly specified in the beginning. Thus, white box checking can be used as a second line of defense against faults that will not be spotted by black box checking.

Most faults in a AC model result from the nondeterministic automata of FSM states, for example, in Fig. 3, white box checking will detect that the value $x$ will result to a ***grant*** of ***access*** when it is either $s$ or $t$. This does not violate the safety requirement, however, the safety property will not be maintained if a more stringent safety requirement requires that only one value of $x$ attribute is desired from the policy.

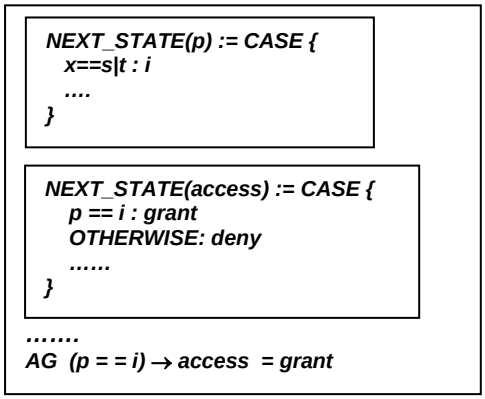


Fig. 3 Example of ambiguous value and the safety requirement in an AC model.

Another example shows a transition to an unspecified state for a certain range of data values such as in Fig. 4, there is no way for the black box checker to figure out the value of ***access*** when $x$ value is other than $s$ unless we check with the safety requirement ***AG ! (p = = i) → access = deny***. This uncovered value can be detected by the white box checking when different values were assigned to $x$, which does not match any expected case condition, and results in the same

***grant*** of ***access***. Thus, the safety requirement verification informs the users which rules are not covered by the existing safety requirement so that the users can add new properties to cover the uncovered.

```
NEXT_STATE(p) := CASE {
   x==s : i
   ....
}

NEXT_STATE(access):= CASE {
   p == i : grant
   ......
}
.......
AG  (p == i) → access  = grant
```

Fig. 4 Example of uncovered value and the safety requirement in an AC model.

### *Coverage and Confinements Semantic faults*

in the safety requirement are completely covered by the model, and to confirm that no exceptional access permissions are granted unless intentionally allowed. The first step of CCC is to discover the rules, which are seeped through the specification of the safety requirement by applying white box checking on mutated versions of the model. The second step is to detect unexpected access permission that might not be the intention of the policy author, by applying model checking on modified rules extracted from the original ones.

*Rule coverage checking*

The key notion of rule coverage checking is to synthesize a version of the given model in such a way that the permission of its rules is mutated such that rule *r* is changed to ~*r*. If safety requirements are satisfied by both mutated and original models through model checking, then some of the rules and their mutants would never be applied to the safety requirements; in other words, the safety requirements do not cover all the rules in the model.

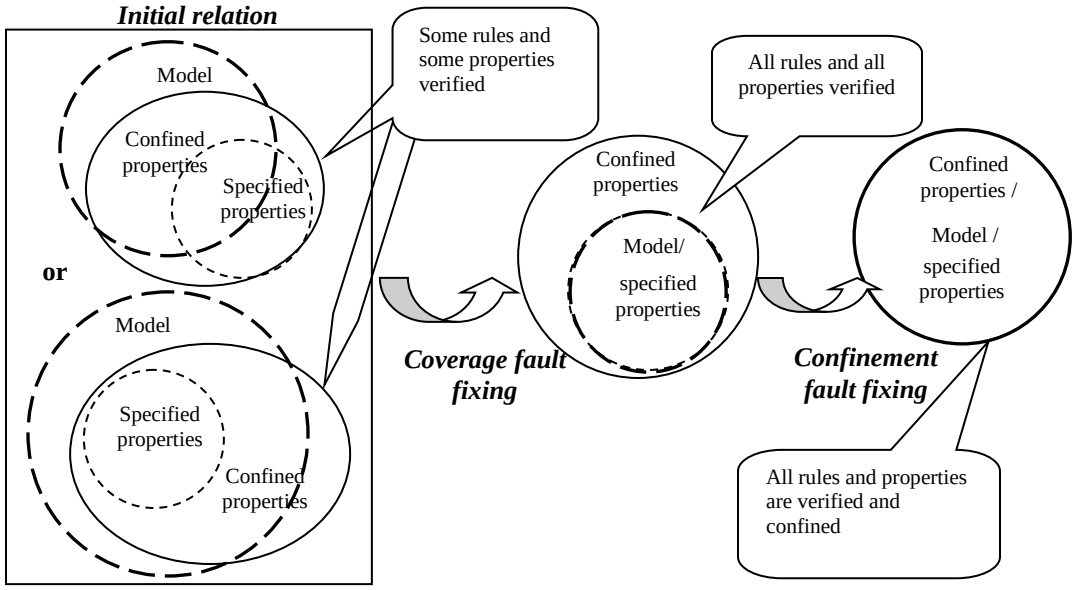


Fig. 5 Relations of Policy, Model, and Safety Requirements

The rules in the policy, model, and safety requirements may each describe their own space of permission conditions, and may not be congruent in one space as the initial relation illustrated examples in Fig. 5. The safety and liveness check can assure only the logical integrity of some rules against some safety requirements. The complete satisfaction of a model to its policy requires fixing of coverage and confinement faults if any violations are detected by additional Coverage and Confinement Checks (CCC), the second line of defense against such semantic faults.

CCC requires mutant versions of the model, and extra modified properties for additional model checking. As illustrated in Fig. 5, the goal of CCC is to ensure that the rules

As an example in Fig. 6, the safety and liveness checking verify that the model conforms the safety requirement ***AG (q = = i) → access = grant*** without counterexamples; however, by applying the CCC by mutating the rule ***u == j: grant*** to ***u == j : deny*** for the coverage checking, the result shows that the safety requirement satisfies the mutated rules as well (without counterexamples), indicating that the variable *u* was never applied to the safety requirement ***AG (q = = i) → access = grant***. This result shows that the rule ***u == j : grant*** is not verified with the property ***AG (q == i) → access = grant***. One way of addressing this insufficiency is adding a new property that describes proper control of *u*. Note that it is necessary to check every *rule in the model* against all safety requirement to achieve thorough verification.

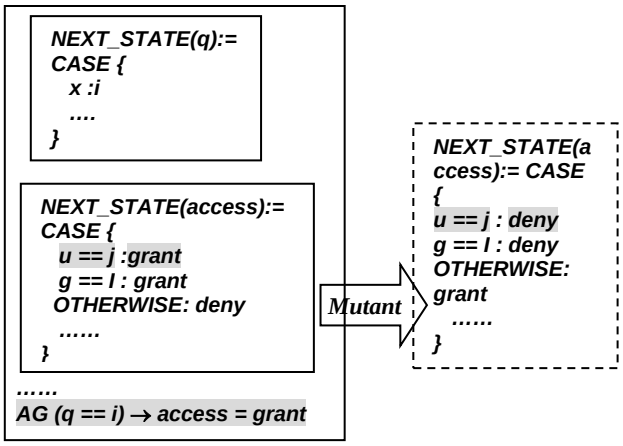


Fig. 6 Example of uncovered rules in a AC model

*Property confinement checking*

Property confinement checking ensures that there is no exceptional permission allowed in addition to the specified safety requirement; this checking requires a modified safety requirement to be added for the next run of model checking. Confinement check should discover the discrepancy of the specified safety requirement and the safety requirement the AC policy author intend. The rationale is that if the model does not satisfy the modified safety requirement, then there are exceptional access permissions that leak through the safety requirement. Fig. 7 shows a transition to an unspecified state for a certain range of data values that allow exceptional permissions not covered by a specified safety requirement because the value of ***access*** when *u* value is different than *i* (such as ***u = j***) also grants access permission by the rule ***otherwise: grant***. This fault can be caught by a counterexample ***AG (u == j) → access = grant*** when checking the model against the additional confinement property ***¬AG (u == i) → access = deny*** derived from original property ***AG (u == i) → access = grant.*** The additional model checking for confinement verification informs the AC policy authors which safety requirement is not confined so that the AC policy author can add new rules to enforce the safety of the model. As in this case, changing the rule ***otherwise: grant*** to ***otherwise : deny*** and adding all granted rules in the state will correct the problem.

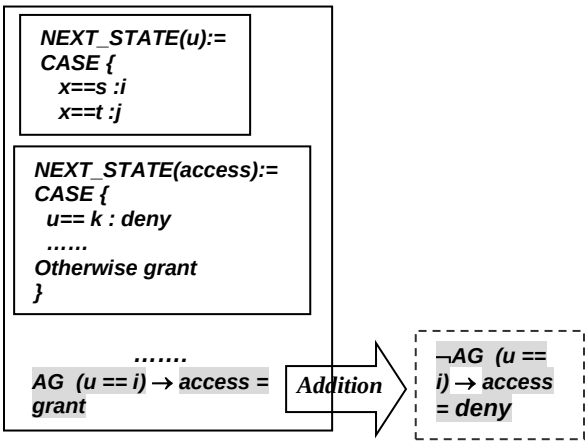


Fig. 7 Unconfined rule in a property

Note that it is possible the AC policy author intentionally allowed the exception for a safety requirement, and it is necessary to check every safety requirement against the set of rules in the model to achieve thorough verification.

***Implementation Test***

Black box model checking and white box mutation test provide methods for verifying the correct model representation of the policy. Once a model is verified, the AC mechanism can be implemented based on the design of the model and additional constraints if needed. Usually AC mechanisms are code developed in a language the AC system supports, for example dedicated AC language such as XACML [7] is commonly used for AC code implementation. AC implementation can be error prone. As the AC model is directly implemented by an algorithm, the errors are often caused by syntactic faults, such as mistakenly changing the + sign to – sign, or typing letter O instead of 0.

The correct implementation of the policy needs to be tested. To achieve that, a test oracle that contains cases of all possible outcome of the AC safety requirement is required, because implementation faults are unpredictable without a logical trace for detecting. Thus, all the combinations of the variables in the safety requirements need to be covered in the oracle. For example, a safety requirement: "*x read y grant*" where *x* has 3 different values and *y* has 5 different values will have 3*2*5*2 (assume that the AC actions has two values: *read* and *write*, and permission has only two values: *grant* and *deny*) test cases in the test oracle. The implemented AC system will then run these test cases to verify whether the actual test outputs are the same as the expected outputs.

It is not uncommon that a verification test includes hundreds of safety requirements; each contains tens of variables, in such case, the number of test cases in a test oracle for the implementation test is too great to be efficiently performed, therefor, additional techniques [8,9] for reducing the test case size without sacrificing the capability may be required for the test.

## V. IMPLEMENTATION CONSIDERATIONS

General AC system testing framework shown in Fig. 8 contains four major functions using the methods as stated in Section IV. The AC Rule Real-time Error detector is used optionally for design the initial AC models [10]. The Black Box Tester checks if a model (original or mutant) holds for the specified Safety Requirements. The Black Box Tester (counterexample results) provides information for original model fix (a human action as dotted line in the Figure) and for mutation killing check for White Box Tester, it also takes output from the Test Generator and returns results for test case generation. The White Box Tester generates and kills mutant models based on the original model and safety requirements; its mutated models are sent to the Black Box Tester for

mutation killing check, or to the AC model author for original model or safety requirement fix (a human action as shown in dotted lines in the figure). The Test Oracle Generator generates test cases based on the Safety Requirement and the Black Box Tester's counterexample results. The process steps are listed below:

1. (optional) AC models is designed based on the AC policy by using AC Rule Real-time Error Checker (as described in Section 5.4).
2. Safety requirements are specified
3. Completed original AC model is checked against safety requirements by Black Box Tester, if an error is found, the original model needs to be fixed (thus repeat steps 1 to 3), otherwise proceed to next step 4.
4. Fixed original model send to White Box Tester for coverage and confinement check. The White Box Tester uses Black Box Tester to decide if generated mutant models were killed. If not, original model or safety requirements need to be fixed (thus repeat step 1 to 4), otherwise, proceeds to next step 5.
5. Test Oracle Generator generates test cases based on Safety Requirements, which are sent to Black Box Tester for generating permission results used for test oracle.

Note that the components in Fig. 8 and steps are not necessarily all required for an AC model verification; the selections of components and steps might depend on the complexity of the model and the cost for implementing the test framework. Thus, an AC model test framework can contain optional components/functions in Fig. 8 except that the Black Box tested is essential.

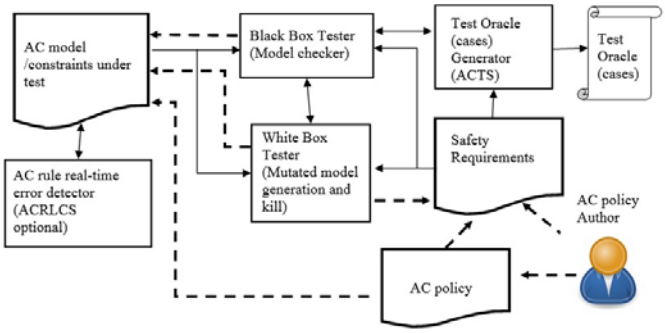


Fig. 8 AC model verification framework

## VI. Related Work

In addition to the FSM based method, other techniques [11] are available for AC model verification such as theorem Proof (including first and higher logic proof) and MTBDD [12] methods.

*Multi-Terminal Binary Decision Diagrams (MTBDD)*

Developed in Racket (formal PLT) Scheme, Margrave [13] is a software tool suite for verifying safety requirements against AC policies written in XACML. Margrave represents XACML policies as MTBDD models, it allows the user to specify various forms of safety requirements in the Scheme programming language. Margrave uses one variable for each attribute-value pair in the XACML policy. Margrave creates MTBDD models for the individual policy rules, then combines these with MTBDD-combining algorithms that implement the XACML rule- and policy-combining algorithms.

Margrave views the policy constants permit and deny as rules; an operation called augment-rule takes a Boolean condition on the variables and a rule and constrains the rule to also require the given condition. It supports query-based verification and provides query-based views by computing exhaustive sets of scenarios that yield different results including change-impact analysis for comparing a pair of policies. Margrave provides the benefits of static verification without requiring authors to write formal properties; its power comes from choosing an appropriate policy model in first-order logic, and embracing both scenario-finding and multi-level policy-reasoning. In general, Margrave identifies formulas corresponding to many common firewall-analysis problems automatically, thus providing exhaustive analysis for richer policies and queries.

*ACPT*

NIST's Access Control Policy Tool (ACPT) [14] provides (1) GUI templates for composing AC models, (2) safety requirements verification for AC models through an SMV (Symbolic Model Verification) model checker NuSMV, (3) complete test cases generated by NIST's combinatorial testing tool ACTS, and (4) XACML policy generation as output of verified model. Through the four major functions, ACPT performs all the syntactic and semantic verifications as well as the interface for composing and combining AC models for AC policies; ACPT is capable of verifying combined policies based on the permission priorities and/or algorithms specified by the user.

ACPT allows users to specify AC models or their combinations, as well as safety requirements through GUI that contains model templates for three major AC policies: static Attribute-Based AC, Multi-Leveled Security, and stated Work-Flow. ACPT then performs black box model check to verify if the specified safety requirements conform to the specified models. If not, non-conformance messages are returned to the user, otherwise, ACPT proceeds to generate test cases through ACTS, which are ready for testing the AC application implemented according to the models.

*Formal Methods*

Formal methods for the validation of access control policies involving mathematical tools and proofs have also been advocated. Rémi Delmas and Thomas Polacsek [15] have proposed a logical modelling framework to find the inconsistencies and incompleteness in access control policies. Providing a mechanism for the detection of these two properties, they have introduced two new properties, applicability and minimality and their proposed technique is capable verifying these two properties [16]. By using the concepts of signatures, formula and predicates, they have defined some rules for the logical framework, which works for limited or finite data so their rules are also applicable to the finite data. They also mentioned that the MSFOL (many-sorted

first order logic) [17] formula should be converted to a pseudo-Boolean logic formula to analyze it. The proposed tool is a three-steps procedure where grounding operation gives the grounded formula in the first step which is converted to a bit-vector expression using the bit-vector encoding in the second step of this process. In the last step of this procedure, the bit-vector expressions are converted into clauses which are in pseudo-Boolean form and give us the pseudo-Boolean formula.

Z [18] is based on axiomatic set theory and first order predicate logic, which can be used for describing and modeling AC policies [19]. Z notation apply set theory forms an adequate basis for building the AC model, which allow syntax and type checking, schema expansion, precondition calculation, domain checking, and general theorem proving for model verification by domain checking. Many of the proof obligations are easily proven. In more difficult cases, generating the proof obligation is often a substantial aid in determining whether a specification in the AC model is meaningful to the AC policy.

## VII. Conclusions

This paper describes a notion of safety for access control, and analyzes verification approaches for static, dynamic, and historical AC models. Static models are those in which no state is retained, while dynamic models may retain state during a session. Historical models include long-term user and object history in access decisions. An AC system is safe if no privilege can be escalated to unauthorized principals, but the correct privileges are always accessible to authorized principals. We also describe a rough taxonomy of faults that may be present access control models.

To verify safety requirements for AC models, we provided a general approach that expresses AC models and AC safety requirements in the formal specification of a black box model or first order logic checkers for verification. Then the black box verifier verifies the specified models against the specified safety requirements. Most of the verification system supports static, dynamic, and historical AC models. In addition to black box checking, white box checking methods make sure that the semantic coverage of the safety requirements also conforms to the intentions of the AC policy authors. Finally, the generation of test cases to check the conformance of the models and their implementations is necessary.

Access Control model and safety requirements conformance verification of generic AC policies bring benefits to society in two aspects. First, it should lead to improved verification practices for testing and verifying AC models in improving AC system quality and security in general. Second, innovations in new testing and verification algorithms and tools tend to propagate quickly across application or task domains where AC policies are used.

## References


[1] V. C. Hu et al, NIST Special Publication 800-162, “Attribute Based Access Control Definition and Consideration.”

[2] V. C. Hu, D. F. Ferraiolo, D. R. Kuhn, NIST Interagency Report 7316, “Assessment of Access Control Systems”.

[3] V. C. Hu, and K. Scarfone, NIST Interagency Report 800-7874 “Guidelines for Access Control System Evaluation Metrics”.

[4] A. Gouglidis, I. Mavridis, V. C. Hu, “Security policy verification for multi-domains in Cloud systems” International Journal of Information Security (IJIS13), article No. s10207-013-0205-x, 13(2), 97-111 in Springer, July, 2014.

[5] Harriosn M. A., Ruzzo W. L., and Ullman J. D., “Protection in Operating Systems”, Communications of the ACM, Volume 19, 1976.

[6] V. Hu, R. Kuhn, T. Xie, and J. Hwang, “Model Checking for Verification of Mandatory Access Control Models and Properties,” International Journal of Software Engineering and Knowledge Engineering (IJSEKE) regular issue IJSEKE Vol. 21, No. 1., 2011.

[7] XACML, https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml

[8] http://csrc.nist.gov/groups/SNS/acts/index.html

[9] V. C. Hu, R. D. Kuhn, T. Xie, “Property Verification for Generic Access Control Models”, in Proceeding of The 2008 IEEE/IFIP International Symposium on Trust, Security and Privacy for Pervasive Application (TSP2008), Shanghai, China, December 17-20 2008.

[10] V. Hu, K. Scarfone, “Real-Time Access Control Rule Fault Detection Using a Simulated Logic Circuit”, Proceeding, 2013 ASE/IEEE International Conference on Privacy, Security, Risk and Trust, Washington D.C., USA September 8th-14th, 2013.

[11] A. Li, Q. Li, V. C. Hu, and J. Di “Evaluating the Capability and Performance of Access Control Policy Verification Tools”, Proceeding The Premier International Conference for Military Communications (MILCOM 2015), Tampa FL, August 17-24, 2015

[12] E. Clarke, M. Fujita, P. McGeer, J. Yang, and X. Zhao, “Multi-terminal binary decision diagrams: An efficient data structure for matrix representation’, International Workshop on Logic Synthesis, 1993.

[13] K. Fisler et al, “Verification and Change Impact Analysis of Access Control Policies”, Proceeding, 27th International Conference on Software Engineering (ICSE’05), Page 196-205, ACM, New York, NY, 2005.

[14] http://csrc.nist.gov/groups/SNS/acpt/index.html.

[15] R. Abassi, S. Fatmi, "An Automated Validation Method for Security Policies: the firewall case", The 4th Int. Conf. on Information Assurance and Security, 2008, pp. 291-294.

[16] M. Aqib, R. A. Shaikh. “Analysis and Comparison of Access Control Policies Validation Mechanisms”, I.J. Computer Network and Information Security, 2015, 1, 54-69.

[17] J. H. Gallier, “Logic for Computer Science: Foundations of Automatic Theorem Proving”, ch. 10, pp. 448‑476, Wiley, 1987.

[18] B. Potter, J. Sinclair, and D. Till, “An Introduction to Formal Specification and Z” Second Edition, by Prentice Hall International Series in Computer Science, 1996.

[19] V. C. Hu, “The Policy Machine For Universal Access Control”, Dissertation, Computer Science Department, University of Idaho, 2002.