NIST Special Publication 800-192

Verification and Test Methods for Access Control Policies/Models

Vincent C. Hu
Rick Kuhn
Dylan Yaga

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-192

C O M P U T E R S E C U R I T Y

National Institute of Standards and Technology
U.S. Department of Commerce

NIST Special Publication 800-192

Verification and Test Methods for Access Control Policies/Models

Vincent C. Hu
Rick Kuhn
Dylan Yaga
Computer Security Division
Information Technology Laboratory

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-192

June 2017

U.S. Department of Commerce
Wilbur L. Ross, Jr., Secretary

National Institute of Standards and Technology
Kent Rochford, Acting NIST Director and Under Secretary of Commerce for Standards and Technology

Authority

This publication has been developed by NIST in accordance with its statutory responsibilities under the Federal Information Security Modernization Act (FISMA) of 2014, 44 U.S.C. § 3551 *et seq.*, Public Law (P.L.) 113-283. NIST is responsible for developing information security standards and guidelines, including minimum requirements for federal information systems, but such standards and guidelines shall not apply to national security systems without the express approval of appropriate federal officials exercising policy authority over such systems. This guideline is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130.

Nothing in this publication should be taken to contradict the standards and guidelines made mandatory and binding on federal agencies by the Secretary of Commerce under statutory authority. Nor should these guidelines be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB, or any other federal official. This publication may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright in the United States. Attribution would, however, be appreciated by NIST.

National Institute of Standards and Technology Special Publication 800-192
Natl. Inst. Stand. Technol. Spec. Publ. 800-192, 68 pages (June 2017)
CODEN: NSPUE2

This publication is available free of charge from:
https://doi.org/10.6028/NIST.SP.800-192

Comments on this publication may be submitted to:

National Institute of Standards and Technology
Attn: Computer Security Division, Information Technology Laboratory
100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930
Email: sp800-192-comments@nist.gov

Reports on Computer Systems Technology

The Information Technology Laboratory (ITL) at the National Institute of Standards and Technology (NIST) promotes the U.S. economy and public welfare by providing technical leadership for the Nation's measurement and standards infrastructure. ITL develops tests, test methods, reference data, proof of concept implementations, and technical analyses to advance the development and productive use of information technology. ITL's responsibilities include the development of management, administrative, technical, and physical standards and guidelines for the cost-effective security and privacy of other than national security-related information in federal information systems. The Special Publication 800-series reports on ITL's research, guidelines, and outreach efforts in information system security, and its collaborative activities with industry, government, and academic organizations.

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; access control testing; access control verification; model testing; policy.

Acknowledgments

The authors Vincent C. Hu, Rick Kuhn and Dylan Yaga of the National Institute of Standards and Technology wish to thank their colleagues who reviewed drafts of this document. The authors also gratefully acknowledge and appreciate the comments and contributions made by government agencies, private organizations, and individuals in providing direction and assistance in the development of this document.

Executive Summary

Access control (AC) systems control which users or processes have access to which resources in a system. They are among the most critical of security components. AC policies are specified to facilitate managing and maintaining AC systems. Faulty policies, misconfigurations, or flaws in software implementations can result in serious vulnerabilities. However, the correct implementation of AC policies by AC mechanisms is a very challenging problem. It is more common that a system's privacy and security are compromised due to the misconfiguration of AC policies rather than the failure of cryptographic primitives or protocols. This problem becomes increasingly severe as software systems become 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.

Thus, AC models are usually written to bridge the gap in abstraction between 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 review methods for the verification for AC models and the testing of model implementations by first defining standardized structures for AC models. We then demonstrate the expressions of AC models and safety requirements in a specification language of a model checker for the use of black box and white box model checkers 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 as well as a method for detecting AC rule faults in real time are discussed.

Table of Contents

List of Figures

List of Tables

1 INTRODUCTION

1.1 Authority

The National Institute of Standards and Technology (NIST) developed this document in furtherance of its statutory responsibilities under the Federal Information Security Management Act (FISMA) of 2002, Public Law 107-347.

NIST is responsible for developing standards and guidelines, including minimum requirements, for providing adequate information security for all agency operations and assets, but such standards and guidelines shall not apply to national security systems. This document is consistent with the requirements of the Office of Management and Budget (OMB) Circular A-130, Section 8b(3), "Securing Agency Information Systems," as analyzed in A-130, Appendix IV: Analysis of Key Sections. Supplemental information is provided in A-130, Appendix III.

This guideline has been prepared for use by federal agencies. It may be used by nongovernmental organizations on a voluntary basis and is not subject to copyright, though attribution is desired.

Nothing in this document should be taken to contradict standards and guidelines made mandatory and binding on federal agencies by the Secretary of Commerce under statutory authority, nor should these guidelines be interpreted as altering or superseding the existing authorities of the Secretary of Commerce, Director of the OMB, or any other federal official.

1.2 Document Scope and Purpose

The purpose of this document is to provide agencies with technical information to assist them in designing, verifying and testing access control (AC) models. The document discusses the principles, requirements, theories, and schemes for the verification and testing of AC models that are designed to assure the AC policy can be implemented by a mechanism.

AC policies are specified to control the access of system resources, AC mechanisms control which users or processes have access to which resources in a system. However, the correct implementations of AC policies by AC mechanisms are very challenging problems. Thus, AC models are usually written to bridge the gap in abstraction between policies and mechanisms to formally and precisely capture the safety requirements that AC systems should adhere to. In this document, we review methods for the verification for AC models and the testing of model implementations by first defining standardized structures for AC models. We then demonstrate the expressions of AC models and safety requirements in a specification language of a model checker for the use of black box and white box model checkers 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, and a method for detecting AC rule faults in real time are discussed.

1.3 Audience and Assumptions

This document is intended to provide practical and conceptual guidance for security managers, administrators, and system testers whose expertise is related to AC. The authors assume that the readers have basic operating system, computing theory, and software assurance expertise, as well as some security expertise, especially in the field of AC. Because of the constantly changing nature of the information technology industry, readers are strongly encouraged to take advantage of other resources (including those listed in this document) for more current and detailed information.

1.4 Document Organization

This document is divided into seven sections, followed by two appendixes. Section 1 states the authority, scope, purpose, audience, and assumptions of this document. Section 2 introduces the general concept of AC policy and model. Section 3 explains the elements of AC safety and faults. The focus of this document is presented in Section 4, which introduces main concepts for AC model verification and testing. Section 5 provides some major methods used to achieve the concepts described in Section 4. Section 6 demonstrates the NIST's implementation of AC verification and test tool: Access Control Policy Tool (ACPT). Section 7 presents the conclusion to the document. Appendixes A - Glossary, B – SOD Policies, and References provide information that supports the document.

2 ACCESS CONTROL POLICIES AND MODELS

The objective of an access control (AC) system is often described in terms of protecting system resources against inappropriate or undesired user access. From a business perspective, this objective could just as well be described in terms of the optimal sharing of information to users and applications. However, a greater degree of sharing may get in the way of resource protection, so a sufficiently fine-grained AC policy should enable selective sharing of information where in its absence, sharing may be considered too risky altogether.

Correct implementation and enforcement of AC policies is based on the premise that the policy specifications are correct without hidden or conflicted rules that cause leaking or blocking access to objects. AC policy specifications must undergo rigorous verification and validation through systematic testing to ensure that the policy specifications truly encapsulate the desires of the policy authors. Verifying and testing of AC policies is a lot like general application software testing, but there are differences as well. To formally and precisely capture the safety requirements that the AC system should adhere to, models are usually written to bridge the rather wide gap in abstraction between policy and mechanism. Thus, an AC model provides unambiguous and precise expression as well as a reference for design and implementation of safety requirements.

2.1 Access Control Policies

AC *policies* are high-level requirements that specify how access is managed and who, under what circumstances, may access what information. While AC policies can be application-specific and thus taken into consideration by the application vendor, policies are just as likely to pertain to user actions within the context of an organizational unit or across organizational boundaries. For instance, policies may pertain to resource usage within or across organizational units or may be based on need-to-know, competence, authority, obligation, or conflict-of-interest factors. Such policies may span multiple computing platforms and applications. [NIST-IR-7316]

It is not practical to generate a list of generic AC policies, since business objectives, tolerance for risk, corporate culture, and the regulatory responsibilities that influence policy differ from enterprise to enterprise, and even from one organizational unit to another. The AC policies within a hospital may pertain to privacy and competency (e.g., only doctors and nurse practitioners may prescribe medication), and hospital policies will differ greatly from those of a military system or a financial institution. Even within a specific business domain, policy will differ from institution to institution. Furthermore, AC policies are dynamic in nature, in that they are likely to change over time in reflection of ever evolving business factors, government regulations, and environmental conditions. There are several well-known AC policies, which can be categorized as discretionary or non-discretionary. Typically, discretionary AC policies are associated with identity-based AC, and non-discretionary ACs are associated with rule-based controls (for example, mandatory security policy).

Discretionary Access Control

Discretionary AC (DAC) leaves a certain amount of AC to the discretion of the object's owner, or anyone else who is authorized to control the object's access. For example, DAC is generally used

to limit a user's access to a file; it is the owner of the file who controls other users' accesses to the file. Only those users specified by the owner may have some combination of read, write, execute, etc. permissions to the file. DAC policy tends to be very flexible and is widely used in commercial and government sectors. However, DAC is known to be inherently weak for two reasons. First, granting read access is transitive; for example, when Ann grants Bob read access to a file, nothing stops Bob from copying the contents of Ann's file to an object that Bob controls. Bob may now grant any other user access to the copy of Ann's file without Ann's knowledge. Second, DAC policy is vulnerable to Trojan horse attacks. Because programs inherit the identity of the invoking user, Bob may, for example, write a program for Ann that, on the surface, performs some useful function, while at the same time destroy the contents of Ann's files. When investigating the problem, the audit files would indicate that Ann destroyed her own files. Thus, formally, the drawbacks of DAC are as follows:

- Information can be copied from one object to another; therefore, there is no real assurance on the flow of information in a system.
- No restrictions apply to the usage of information when the user has received it.
- The privileges for accessing objects are decided by the owner of the object, rather than through a system-wide policy that reflects the organization's safety requirements.

ACLs and owner/group/other AC mechanisms (e.g. permission bits in UNIX) are by far the most common mechanisms for implementing DAC policies. Other mechanisms, even though not designed with DAC in mind, may have the capabilities to implement a DAC policy.

Non-Discretionary Access Control

In general, all AC policies other than DAC are grouped under the category of non-discretionary AC (NDAC). As the name implies, policies in this category have rules that are not established at the discretion of the user. Non-discretionary policies establish controls that cannot be changed by users, but only through administrative action. Static NDAC, for example, are MLS, ABAC, and RBAC, and dynamic NDAC such as Separation of duty (SOD) policy can be used to enforce constraints on the assignment of users to roles or tasks. An example of a constraint is the requirement that two roles be mutually exclusive; e.g., if one role requests expenditures and another approves them, the organization may prohibit the same user from being assigned to both roles. So, membership in one role may prevent the user from being a member of one or more other roles, depending on the SOD rules, such as Work Flow and RBAC. Another example of NDAC is a history-based SOD policy that regulates, for example, whether the same subject (role) can access the same object for a variable number of times.

Due to the fact that DAC policies have no well-defined variables in terms of model definition, they are in general hard to formalize and verify efficiently with first order models, which most of the black box and white box verification tools are based on. This document does not cover DAC policies.

2.2 AC Models

Rather than attempting to evaluate and analyze AC policies exclusively at the mechanism level, security models are usually written to describe the security properties of an AC policy. An AC *model* is a formal presentation of an AC policy enforced by the mechanism and is useful for proving theoretical limitations of a 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 specification of 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 of a wide variety of policies and policy classes. In general, non-discretionary AC polices can be modeled by static, dynamic and historical Finite State Machine (FSM) models from one of the three following classes: **static**, **dynamic** and **historical**.

a) Static Policy Class

Policies in the Static Policies Class 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 [NIST_SP-162], MLS, and RBAC [NIST-IR-7316]. 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 Policy Class

Policies in the Dynamic Policy Class may include temporal constraints that regulate access permission 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 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 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 Policy Class

Policies in the Historical Policy Class regulate access permissions by historical access states or recorded and predefined series of events. Representative AC policies for this type of AC policies include Chinese Wall and Workflow AC policies. This policy class can be best described by **synchronous** or **direct specification** expressions of an FSM model. For example, the synchronous FSM specification in Example 3 specifies a Chinese Wall AC policy where there are two Conflict of Interest groups COI_1, COI_2 of objects:

{ VARIABLES
access {grant, deny};
act {rd, wrt};
object {none, COI_1, COI_2};
state {1, 2, 3};
INITIAL_STATE(state) := 1;
INITIAL_STATE(object) := none;
NEXT_STATE(state) := CASE {
state == 1 & act == rd & object == COI_1: 2;
state == 1 & act == rd & object == COI_2: 3;
state == 2 & act == rd & object == COI_1: 2;
state == 2 & act == rd & object == COI_2: 2;
state == 3 & act == rd & object == COI_1: 3;
state == 3 & act == rd & object == COI_2: 3;
OTHERWISE: 1; };
NEXT_STATE(access) := CASE {
state == 2 & act == rd & object == COI_1: grant;
state == 3 & act == rd & object == COI_2: grant;
OTHERWISE: deny; };
NEXT_STATE (act) := act;
NEXT_STATE (object) := 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. 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 for inconsistency and incompleteness of the model and to verify safety requirements, because an AC model does not necessarily explicitly express the policy, which can also be implicitly embedded by mixing with direct access constraints or other AC models.

3 ACCESS CONTROL SAFETY AND FAULTS

It is more common for a system's privacy and security to be compromised due to the faulty AC models and mechanism of AC policies than by the failure of cryptographic primitives or protocols. AC faults include leaking privileges, blocking authorized access, and failure to reach the correct result in grant/deny decisions. Such faults can result in serious vulnerabilities, especially when different AC models and constraint rules are combined into one model. This problem becomes increasingly severe as systems become more complex, and are deployed to manage a large amount of sensitive information and resources that are organized into sophisticated structures.

3.1 Safety

Safety is the fundamental property of an AC system, which ensures that the AC model will not result in the leakage of permissions to an unauthorized principal. Thus, an AC model is said to be 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 [NIST-IR-7874].

Among all the safety features, Separation of Duties (SoD) (see Appendix B) is 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 executed by the same user) or dynamically (by enforcing the control at access time).

3.2 AC Faults

AC faults compromise the safety as described in 3.1, at a 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 a 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 an 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 an 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 where a rule has a dependency on other rule(s), which eventually depend back on the rule itself, so 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 users(groups), which also in a chain of inheritance relation inherit back to the user(group)'s privilege. For example, user *x* inherits privilege from user *y*, which inherits privilege from user *z*, which in turn inherits privilege from user *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.

3.3 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 systems 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, Figure 1 shows that privilege *k* inherits privilege *j* through legal inter-system privilege inheritance (because both have 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 policy, which should be preserved during collaborations. [GMH14] A meta policy is a policy that is usually applied for reconciling policy conflicts or to handle priorities of access decisions rendered from more than one policy. Thus, in addition to autonomy requirements, AC safety requirement may include a priority model within the meta policy.

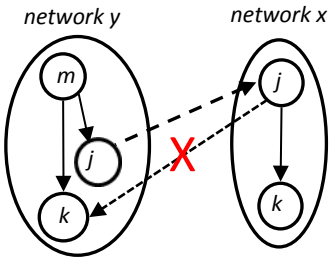


Figure 1 Privilege leaks through inter-system privilege inheritance

3.4 Vulnerability Hierarchies in Access Control Configurations

It has been shown that there is a hierarchical relationship among vulnerabilities in access control systems [Kuhn11], such that the conditions that allow the exploitation of one are sufficient for triggering other vulnerabilities downstream in the hierarchy. While the number of potential flaws that result in vulnerabilities is vast, the structure of access control rules results in a hierarchy for certain classes of vulnerabilities. This section explains these hierarchies and how they may be used to reduce the number of tests required.

Access Control Rule Structures

An access control policy, *P*, is implemented by a set of rules, *R*. A vulnerability can be defined as a condition under which the decision from rules *R* differs from the intended result specified by policy *P*, that is, where $R \neq P$. At a high level, vulnerabilities may be grouped into classes of unauthorized access, (possibly partial) denial of service, or a combination of the two, depending on the conditions under which the decision is *grant* or *deny*. For rules where the decision is *grant*, there are three vulnerability possibilities, where R_c = *grant* conditions in implemented policy and P_c = *grant* conditions in correct policy:

1. $R_c \Rightarrow P_c$: a (possibly partial) denial of service (because $R_c \subset P_c$).
2. $R_c \Leftarrow P_c$: unauthorized access (because $R_c \supset P_c$)
3. *otherwise* : possible combination of unauthorized access or denial of service

In cases where the decision is *deny*, the situation is the mirror image of that for *grant* rules:

1. $R_c \Rightarrow P_c$: unauthorized access (because $R_c \subset P_c$)
2. $R_c \Leftarrow P_c$: a (possibly partial) denial of service (because $R_c \supset P_c$).
3. *otherwise* : possible combination of unauthorized access or denial of service

An access control policy rule structure is defined here as a configuration of logical operators, policy terms, and decisions, categorized according to how the rules are constructed. Each rule has a condition and a decision. Conditions may be composed of other conditions connected by logical operators, often with a standard structure. Decisions are often binary, but may also have three or more values (e.g., *grant*, *deny*, *defer*). Methods of rule combining may also have a pattern. In many

cases, a set of rules that can lead to a grant decision are followed by a single default case "else deny". Alternatively, rules with grant or deny decisions may be intermixed, followed by a default case. Thus, one way to categorize access control policy rule structures is to specify the condition format, number of decision values, and rule combining format. This taxonomy is introduced solely for the purpose of characterizing access control structures for which fault hierarchies are developed.

A taxonomy has been defined in [Kuhn11] with the following rule structure attributes and values:

- Condition format: *con* – conjunction of conditions; *dis* – disjunction of conditions; *cnf* – conjunctive normal form of conditions.
- Number of decision values: binary or *n*-ary .
- Rule combining: singular (all rule decisions of the same type, e.g., *grant* or *deny*, followed by a default), or mixed, which refers to mixed grant and deny decisions in rules.

Structures can then be categorized in the format [condition format]/[number of decision values]/[rule combining method]. Some structures that can be defined using this taxonomy are discussed below.

A. *con*/2/*singular*: Decisions are determined by the conjunction of conditions under which access is granted. If no 'grant' decision matches the input configuration, access is denied.

B. *con*/2/*mixed*: Decisions are determined by the conjunction of conditions under which access is granted or denied. If no grant decision matches the input configuration, access is denied. The rule set is as defined for *con*/2/*singular* except that decisions above the default may be either grant or deny.

C. *disj*/2/*singular*: Decisions are determined by a disjunction of conditions under which access is granted. If no 'grant' decision matches the input configuration, access is denied.

D. *disj*/2/*mixed*: Decisions are determined by the disjunction of conditions under which access is granted or denied. If no grant decision matches the input configuration, access is denied. The rule set is as defined for disj/2/singular except that decisions above the default may be either grant or deny.

E. *cnf*/2/*singular*: Decisions are determined by conditions in conjunctive normal form (CNF) under which access is granted. If no 'grant' decision matches the input configuration, access is denied. This rule class is included because it is relatively common in real-world access control problems. The XACML standard is one such widely used framework for implementing access control policies.

F. *cnf*/2/*mixed*: Decisions are determined by the conjunction of conditions under which access is granted or denied. If no grant decision matches the input configuration, access is denied. The rule set is as defined for *cnf*/2/*singular* except that decisions above the default may be either grant or deny.

G. [*condition*]/*n*/[*combining*] (*n*-ary Decision Rules) When possible decisions are more than grant or deny, rule structures are defined as above with the modification that grant decisions

Vulnerability Classes

This section defines classes of vulnerabilities and determines vulnerability hierarchies for the access control policy structures defined previously. A large number of vulnerabilities can be defined for access control rules, but a reasonable set may include the following. Note that the exact form will vary with the type of access rule structure as defined in the taxonomy above.

1. Add Condition: A condition *ca* that was not specified has been added to the implementation:
2. Delete Condition: A specified condition is missing from the implementation:
3. Replace Condition: A condition is replaced with a different one, not equivalent:
4. Stuck at True Condition: A condition is always true:
5. Stuck at False Condition: A condition is always false:
6. Negate Condition: A condition is negated:
7. Negate Decision: The specified result for a condition is the opposite of intended. This mutation can apply only when there are two possible decisions, generally Grant or Deny:
8. Delete Rule: A specified rule has been omitted from the implementation.

In practical implementations, some of these vulnerabilities could arise from administrator error, such as accidentally deleting or leaving out a condition or rule, and others may result from software failures, such as a module that is intended to verify an attribute and return true or false, but always returns true.

Vulnerability Hierarchies

For the eight vulnerability classes introduced above, hierarchical relationships exist for the access control rule structures in the taxonomy. (For proofs and additional discussion see [Kuhn11].) In the structures below, vulnerabilities are abbreviated as follows:

1. *ac*: add condition
2. *dc*: delete condition
3. *rc*: replace condition
4. *tc*: true condition
5. *fc*: false condition
6. *nc*: negate condition
7. *nd*: negate decision
8. *dr*: delete rule

Note that this set of vulnerabilities is not claimed to be complete. It could be extended, for example, with vulnerabilities such as “add rule” or “replace rule”. Hierarchical relationships among detection conditions for different vulnerability classes can be determined by checking for implications between the pairs of detection conditions. That is, for all pairs of vulnerabilities P_i , P_j , $i \neq j$, compute $P \oplus P_i \Rightarrow P \oplus P_j$, where P_i and P_j are access rules for particular faults that lead to vulnerabilities, where $\oplus$ designates XOR.

Hierarchies can be shown to exist for policies with multiple (two or more) rules with Grant decisions, where each rule contains two or more conditions that can be treated as Boolean terms, and the same condition is affected in the correct and faulty policy. The analysis below is for the common case of policies containing a series of rules with Grant decisions, followed by a default Deny decision, where each rule has a single variable or condition, or a conjunction of two or more conditions, as shown below.

if ($c_{11} \cdot \ldots \cdot c_{1n1}$) then grant;
if ($c_{21} \cdot \ldots \cdot c_{2n2}$) then grant;
. . .
if ($c_{k1} \cdot \ldots \cdot c_{kn3}$) then grant;
else deny;

(Similar hierarchies can be constructed for other rule patterns in the taxonomy defined above, but are not shown here.)

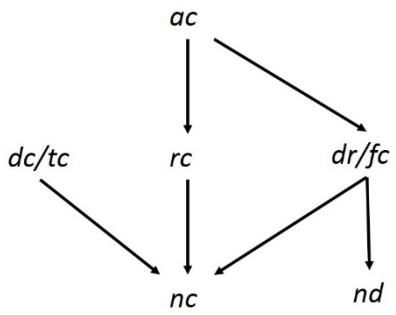


Figure 2 Example Vulnerability Hierarchy

Figure 2 shows an example of vulnerability hierarchy, note that for these policies, *dc* and *tc* are equivalent, because setting one term to *true* in a set of terms in a conjunction is equivalent to deleting the term, e.g., setting *a* to *true*, $abc = 1bc = bc$. Similarly, setting a term to false for this class of rules is equivalent to deleting the entire rule, e.g., setting *a* to *false* , $abc \rightarrow G$ becomes $0bc \rightarrow G = 1 + G = 1$.

Implications for testing

The hierarchy shows that tests capable of identifying certain faults will also identify other types of faults that are subsumed by the former in the hierarchy. As a result, we can determine that some tests are redundant if they are designed to test two types that are related in the hierarchy. Similarly, if mutation testing is being used, the hierarchy tells us in advance which mutations will be equivalent, without generating mutants and applying a model checker or other tool for analyzing the logic.

These concepts can be clarified with an example. Note that in the hierarchy at *stuck at true condition, tc*, subsumes a *negated condition, nc*. A hypothetical policy can be defined as below:

if (a & b & c) then *grant*; if (d & e) then *grant*; else *deny*;

modeled by:

$$P = ((a\ b\ c) \rightarrow G)\ ((d\ e) \rightarrow G) \cdot (\sim(a\ b\ c) \cdot \sim(d\ e) \rightarrow \sim G)$$

Suppose the policy is implemented incorrectly, leaving out condition a:

$$P_{tc} = ((b\ c) \rightarrow G)\ ((d\ e) \rightarrow G) \cdot (\sim(b\ c) \cdot \sim(d\ e) \rightarrow \sim G)$$

The detection conditions for this type of flaw in P_{tc} are:

$$P \oplus P_{tc} = \bar{a}\ b\ c\ \bar{d} + \bar{a}\ b\ c\ \bar{e}$$

That is, for inputs of either $\bar{a}\ b\ c\ \bar{d}$ or $\bar{a}\ b\ c\ \bar{e}$, policy P_{tc} produces an incorrect result, so a test with either of these inputs will detect the error. A different faulty policy may replace the condition in rule 1 with its negation:

$$P_{nc} = ((\sim a\ b\ c) \rightarrow G)\ ((d\ e) \rightarrow G)\ (\sim(\sim a\ b\ c) \cdot \sim(d\ e) \rightarrow \sim G)$$

For the implementation P_{nc}, the detection conditions are:

$$P \oplus P_{nc} = b\ c\ \bar{d} + b\ c\ \bar{e}$$

Thus a test with $b\ c\ \bar{d} + b\ c\ \bar{e}$ (and either a or $\bar{a}$) would detect the error. Notice that $P \oplus P_{tc}, \Rightarrow P \oplus P_{nc}$, i.e., the detection conditions for P_{tc} subsume those for P_{nc}, so a test that detects the faulty policy P_{tc}, with the true condition, will also detect policies with the same condition negated in the same rule. For example, the test $\bar{a}\ b\ c\ \bar{d}$ would detect both vulnerabilities by evaluating to G in both faulty implementations, instead of the correct result $\bar{G}$.

4 POLICY VERIFICATION PRINCIPLES

Identifying discrepancies between AC policy, model, and implementation is crucial because correct enforcement of policies is based on the premise that the policy specifications and implementations are correct, therefore the policy specification and implementation must undergo rigorous verification and validation through systematic verification and testing to ensure that they truly encapsulate the desired AC properties from the policy authors. Note that unlike statements of general programming language, where the later value assignment can overwrite the previous ones of the same variable in the sequence of programming logic, privilege assignment in AC rules has no priority orders, i.e. privilege assignment should not be overwritten by the later ones. Therefore, the fundamental goal of AC policy and implementation verification is to detect the conflicting or missing rules (i.e. policy statements) by verifying the 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 [HRU76] for discretionary AC policy models, which are impossible to be described by static policies, practical safety constraints such as confinements can be specified for discretionary AC policies. As a result, verifications can be performed upon the constraints.

Safety requirements

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 blueprint 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 separation of duty (SoD) property, a safety requirement may specify that 1) subject *x* and *y* are mutually exclusive if neither one inherits the other's privilege directly or indirectly, 2) If subject *x* and *y* are mutually exclusive, then 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. (Antonios)

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. Ensuring the conformance of a model to the policy in principle is to formally detect inconsistency and incompleteness faults as described in Section 3. 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.

4.1 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 confidence in 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. [HKXH11].

In terms of AC attributes, the formal definition of AC model can be illustrated by a deterministic finite state transducer corresponding to a Finite State Machine (FSM) with a five-tuple $M = (\Sigma, ST, s_0, \delta, F)$, where: Σ 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, δ 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 Section 2.2 a), the FSM M_{static} does not require internal states *st* 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 Section 2.2 b), the input alphabets of FSM $M_{dynamic}$ are $\Sigma_{dynamic} = \{gCond_1, \ldots, gCond_n\}$, where $gCond_i$ 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, or the maximum number of accesses allowed for a Limited_Number_of_Access policy. For **historical** AC models as described in Section 2.2 c), the input alphabets of the FSM $M_{historical}$ are $\Sigma_{historical} = \Sigma - \{gCond_1, \ldots, gCond_n\}$, where $sCond_i$, $aCond_i$, and $oCond_i$ contribute to a historical recording 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 $p = (s_i * sCond_1 * \ldots * sCond_n * aCond_1 * \ldots * aCond_n * oCond_1 * \ldots * oCond_n * gCond_1 * \ldots * gCond_n) \rightarrow d$, where $p \in P$ 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 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. Figure 3 shows the relations between M and P in a model checking framework.

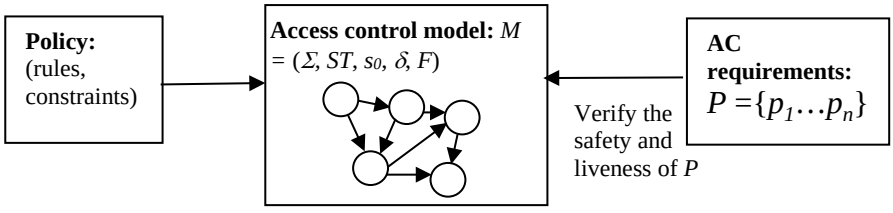
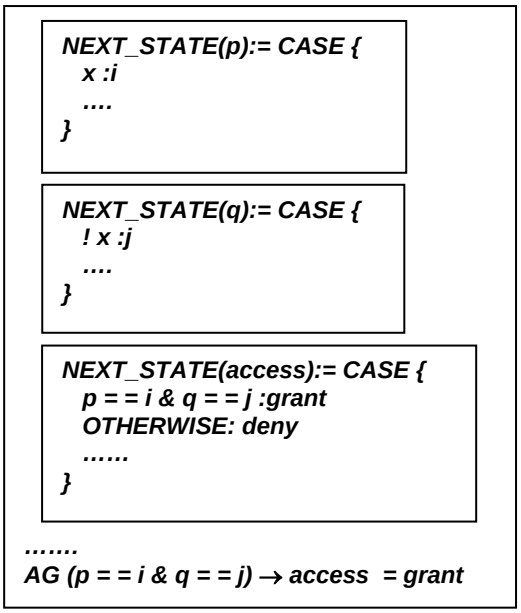


Figure 3 AC model and safety requirement

Thus, the AC rules define the system behaviors that function as the transition relation δ 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 $M \models Ab \rightarrow AXp$, where temporal logic quantifier *A* represents "always", and logic quantifier *X* represents "is true next state". The purpose of safety verification 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 requirement.

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 Figure 4, 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.



AG means that condition ***(p = = i & q = = j) → access = grant*** is always, or globally, true in all the states of all the possible path in the model by CTL check

Figure 4 Example of unreachable state in the model

To detect this kind of semantic fault, 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 [MX07] 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 Figure 4, 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. 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 an AC model result from the nondeterministic automata of FSM states, for example, in Figure 5, 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 meet be maintained if a more stringent safety requirement requires that only one value of x attribute is desired from the policy.

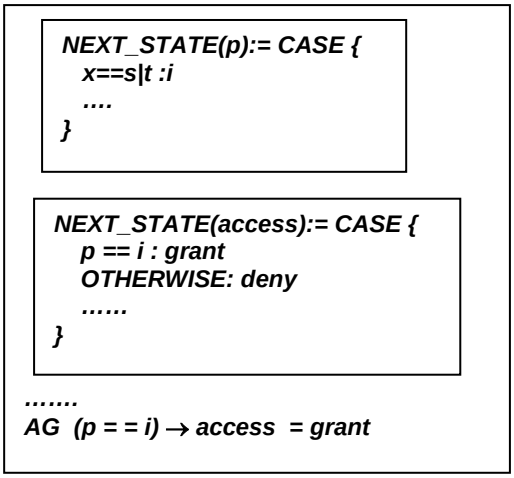


Figure 5 Example of ambiguous value and the safety requirement in an AC model.

Another example shows a transition to an unspecified state for a certain range of data values such as in Figure 6. There is no way for the black box checker to determine 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 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.

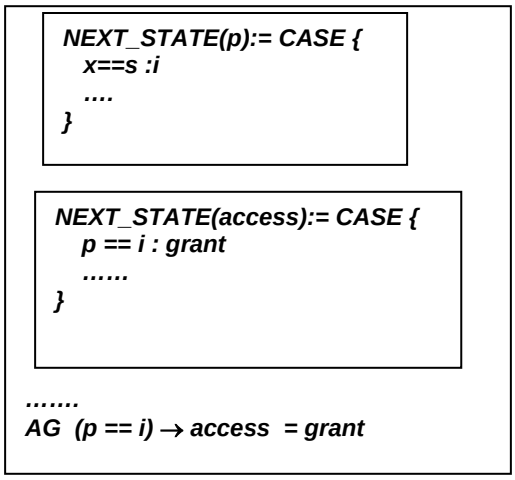


Figure 6 Example of uncovered value and the safety requirement in an AC model

4.2 Coverage and Confinements Verifications

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 Figure 7. The **safety** and **liveness** check can assure only the logic integrity of some rules against some safety requirements. The complete satisfaction of a model to its policy requires repair of **coverage** and **confinement** faults if any violations are detected by additional Coverage and Confinement Check (CCC), the second line of defense against such semantic faults.

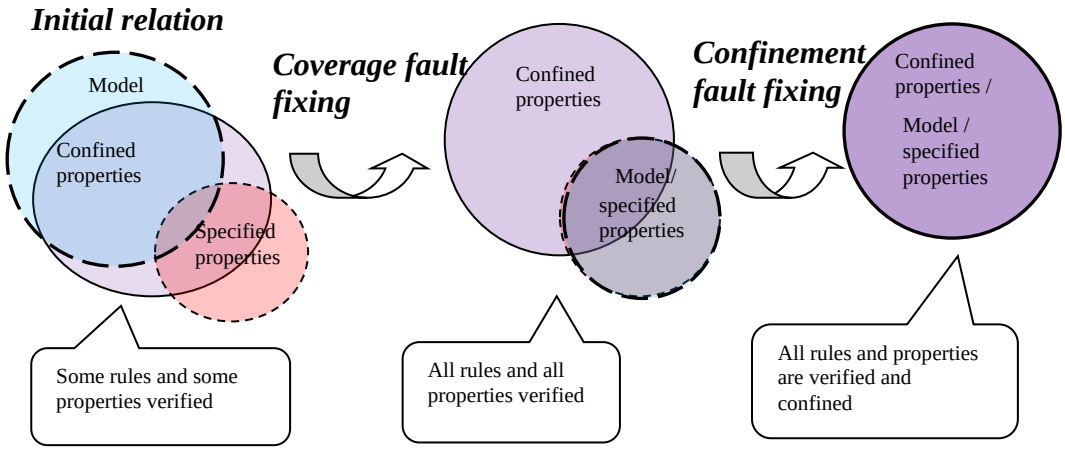


Figure 7 Relations of Policy, Model, and Safety requirements

CCC requires mutant versions of the model, and extra modified properties for additional model checking. As illustrated in Figure 7, the goal of CCC is to ensure that the rules 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.

As an example in Figure 8, 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***.

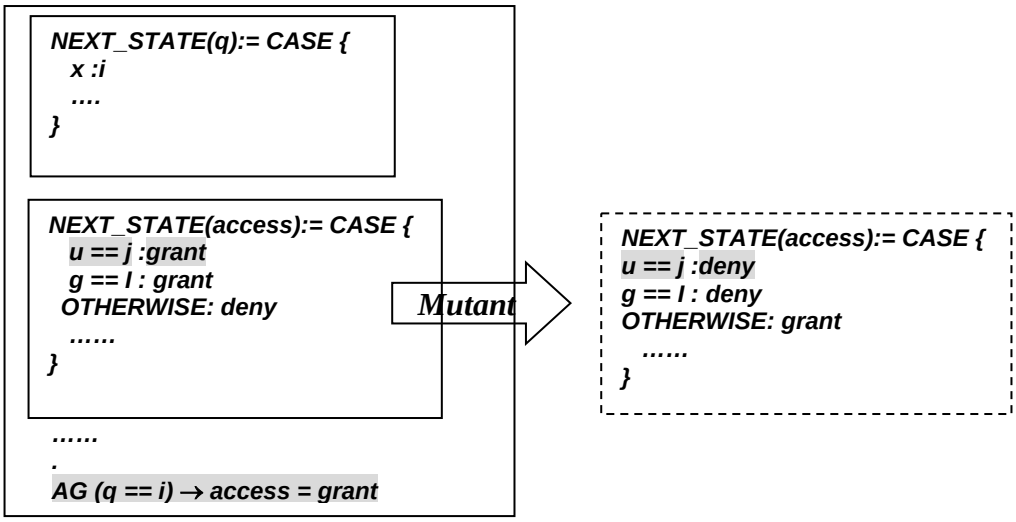


Figure 8 Example of uncovered rules in a AC model

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 by 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.

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.

Figure 9 shows a transition to an unspecified state for a certain range of data values that allow exceptional permission 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.

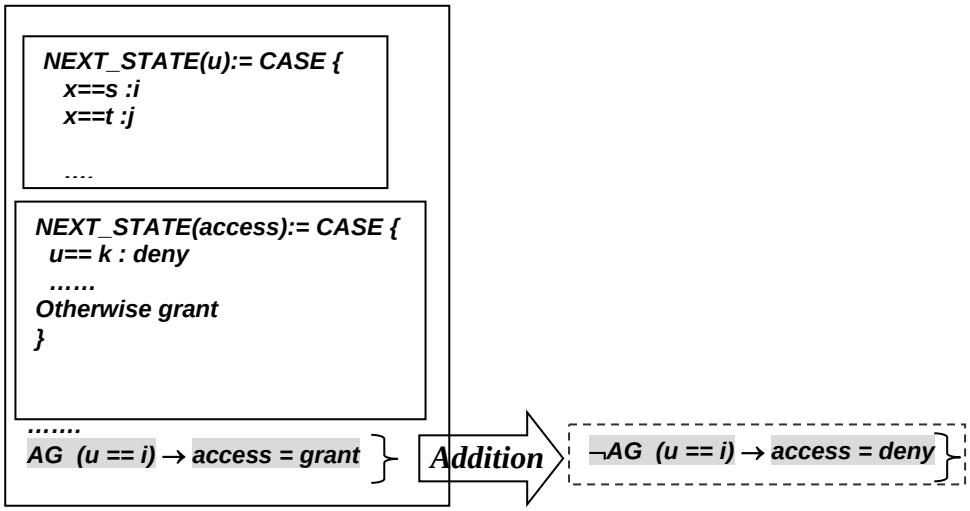


Figure 9 Unconfined rule in a property

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.

4.3 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 [XACML] 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 a 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 outcomes 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 then there will be $3 \cdot 2 \cdot 5 \cdot 2 = 60$ test cases (assume that the AC actions has two values: *read* and *write*, and permission has only two values: *grant* and *deny*). 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 [ACTS] [HKX08] for reducing the test case size without sacrificing the capability may be required for the test. Section 5.6 describes combinatorial methods for making such large problems tractable.

5 VERIFICATION AND TESTING TECHNOLOGY

To verify the correctness of an AC model, the model and verified safety requirements are specified in a formal logic formula (including first and higher logic proof). For example, safety requirements against the model are verified using a model checker. In this process, confidence in the model's correctness depends on the quality of the specified safety requirements. Hence, white box analysis can be applied to check if the entities (i.e., rules) of the model are sufficiently covered and confined by the safety requirements, as well as the quality of given safety requirements. Finally, test cases are generated from the domain variables in the AC model and specified safety requirements. These test inputs are fed into real AC implementation of the given model to verify whether the actual test outputs are the same as the expected outputs. (IJEKE)

5.1 Model Checker

A model checker provides the modular hierarchical descriptions, and for the definition of reusable components of the specification of 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. 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 (as described in Section 2.2) of FSM expressions for specifying AC models in terms of the sequence of the state transitions:

- Synchronous. An AC model is expressed by defining the value of AC constraints in the next state (i.e., after each transition), given the value of constraints in the current states (i.e., before transition).

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

- Direct specification. An AC 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.

FSM Model Checking Basic

Based on the type of transition state (as describe above) of the FSM specification, a model checker verifies if a model satisfies a set of desired safety requirements specified by the user. The specifications can be checked by temporal logics such as Computation Tree Logic (CTL), and Linear Temporal Logic (LTL) extended with Past Operators. CTL and TTL formulas consist of

atomic propositional logic formulas, plus temporal connectives. The propositional logic formulas are expressions about the state of the system. The temporal connectives are expressions about paths into the future that the state of the system can follow [NuSMV]. And the model checker constructs and prints a counterexample, i.e. a trace of the FSM that falsifies the property.

CTL is a branching-time logic that takes into account the non-deterministic, branching evolution of a FSM. More precisely, the evolution of a FSM such that from a given state the evolution of FSM can be described as an infinite tree, where the nodes are the states and the branching represents the non-determinism transitions of the states. The paths in the tree from one state to the others are the possible evolutions of the FSM from that state. In CTL, one can express properties that should hold for all the paths that start in a state, as well as for properties that should hold just for some of the paths.

LTL shares a significant intersection that includes most of the common properties used in CTL; it characterizes each linear path induced by the FSM (linear-time approach). The main difference between LTL and CTL formulas is the absence of existential and universal operators, thus LTL temporal operators do not have path quantifiers. In fact, LTL formulas are evaluated on linear paths, and a formula is considered true in a given state if it is true for all the paths starting in that state, which reflect the linear-time paradigm. LTL properties can also include past temporal operators.

CTL and LTL operators can be nested in an arbitrary way and can be combined using logic operators. Typical examples of CTL formulas are *AG !p* ("condition *p* is absent in all the evolutions"), *AGEF p* ("it is always possible to reach a state where *p* holds"), and *AG* ($p \rightarrow AF\ q$) ("each occurrence of condition *p* is followed by an occurrence of condition *q*"). And typical example of LTL is *G p*, stating that condition *p* holds in all future time instants, and *F p*, stating that condition *p* holds in one of the future time instants.

A model checker can also find a counterexample of increasing length, and immediately stops when it succeeds, declaring that the formula is false. The maximum number of iterations can be assigned such that if the maximum number of iterations is reached and no counterexample is found, then the checking is terminated without conclusion that the formula is true, except that any counterexample should be longer than the maximum length.

In general, Bounded Model Checking (BMC) can find two kinds of counterexamples, depending on the property being analyzed. For safety requirements with the bounded number of transitions, a counterexample is a finite sequence of transitions through different states. For liveness properties, counterexamples are infinite but periodic sequences, and can be represented in a bounded setting as a finite prefix followed by a loop, i.e. a finite sequence of states ending with a loop back to some previous state. So, a counterexample demonstrates the falsity that a safety requirement cannot be a finite sequence of transitions.

AC Model Specification in a Model Checker

From the viewpoint of model checking, the satisfaction of an AC model to the safety requirements is composed of two requirements. These requirements can be specified as follows in model checking terms.

(1) **Safety** means that the AC model satisfies the specified safety requirements. Implicit in this description of safety is that there is no violation of the constraints specified in the safety requirements and it is assured that the model will eventually be in the desired situation after it took actions in compliance with the model.
(2) **Liveness** means that the AC model will have neither a deadlock in which the model waits forever for AC events, nor a livelock in which the model repeatedly executes the same operations forever.

Figure 10 describes the relations between the AC model, safety requirements, and the model checking framework. The structure model defines in the Kripke structure $M = (\Sigma, ST, s_0, \delta, F)$ as described in Section 4.1, and safety requirements are described by a set of access rules and system entities. The policy will then be translated to a set of atomic propositions whose truth values are true in the situations. These atomic propositions can represent the assignment of the parameters in each situation.

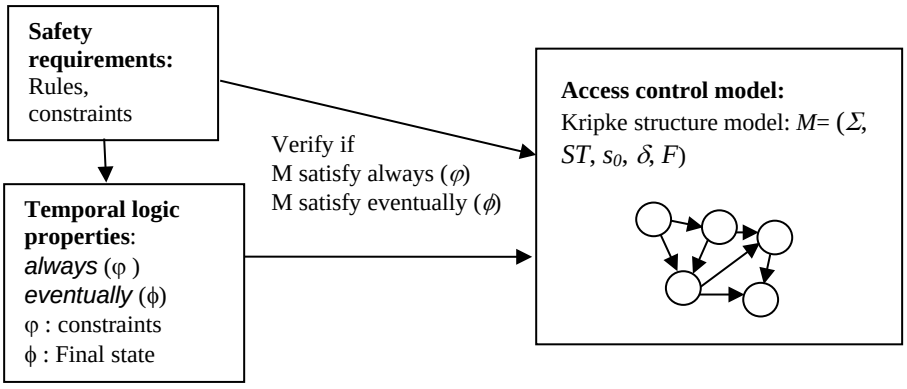


Figure 10 AC model and property

The AC rule defines the model behavior that functions as the transition relation $M = (\Sigma, ST, s_0, \delta, F)$. Then when the constraint is represented by CTL formula φ, we can represent the assertion (safety requirement) that model M always satisfies φ by $M \models A\,\varphi$ using temporal logic operator A to represent "always". Likewise, when the final state is represented by CTL formula ϕ, the assertion (safety requirements) that model M eventually results in the state satisfying ϕ can be represented by $M \models E\phi$ using temporal logic operator E to represent "eventually". The purpose of verification and validation using model checking is to determine whether these assertions (safety requirements) are true, and to identify the state in which the assertions (safety requirements) are not true as a counterexample. Since the behavior of the model can be represented by the Kripke structure, the safety requirement that the model must satisfy can be represented by temporal logic formulas, as shown in Figure 10. Using the preceding definition, we can define the validity of

safety requirements more precisely as the safety requirement is valid to the model if the model can be lead from every possible state that is reachable from initial states to the defined final state while complying with the operation rule and without violating given constraints [KTAK07].

The state transitions of AC models can be categorized by three types of constraints, namely static, dynamic, and historical constraints. These constraints can be expressed directly by the three basic types of FSM models. Thus, an AC model and its safety requirements can be specified in a model checking environment as the examples below:

For static model Example 1 in Section 2.2, specifications of static constraints are verified through the following safety requirements example expressed in 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 should be granted, and all non-compliant ones should be denied. The form "$AG\ (b) \rightarrow AX\ (d)$" indicate that for all paths (the "*A*" in "*AG*") for all states globally (the "*G*"), if *b* holds then ("→") for all paths, in the next state (the "*X*" in "*AX*") *d* will hold.

For dynamic model Example 2 in Section 2.2, specifications of dynamic constraints are verified through the following safety requirements example expressed in temporal logic formula:

AG (state == entering) & (act == rd) & (object == obj) → AX (access = grant)
AG (state == idle | state == critical | state == exiting) → AX (access = deny)

where temporal logic formula $AG\ (b) \rightarrow AX(d)$ indicates that "if *state* is true in condition *b*, condition *d* is true at all times later than the *state*.

For historical model Example 3 in Section 2.2, specifications of historical constraints are verified through the following safety requirements example expressed in temporal logic formula:

AG ((u_state == 2 & act == rd & o_state == COI$_1$) | (u_state == 3 & act == rd & o_state == COI2)) → AX (access = grant)
AG ! ((u_state == 2 & act == rd & o_state == COI$_1$) | (u_state == 3 & act == rd & o_state == COI2)) → AX (access = deny)

temporal logic $AG(b) \rightarrow AX(d)$ indicates that the access event *d* is invocated by historical events in *b*.

Model Checking methods implemented in NuSMV [NuSMV-M] is used by some AC policy verification tools such as NIST's ACPT and MOHAWK [MOHAWK].

5.2 Multi-Terminal Binary Decision Diagrams (MTBDD)

Underlying representation of access-control policies, Multi-Terminal Binary Decision Diagrams (MTBDDs) [CFMYZ93] are a form of decision diagram that map bit vectors over a set of variables

to a finite set of results. Figure 11 shows an example of a policy MTBDD representing a simple security policy in which *faculty* (*f*) can *assign* (*a*) *grades* (*g*) and *students* (*s*) can *receive* (*r*) *grades* (*g*). The policy MTBDD has five variables (*f, s, r, a,* and *g*). Each combination of Boolean values over these variables maps to one of three policy results (*permit*, *deny*, or *not-applicable*); the results are denoted by the terminals of the policy MTBDD. Given an assignment of Boolean values to the variables, traversing a policy MTBDD from the root to a terminal according to the variable values indicates the result of the policy under that assignment. [Fisler et al 05].

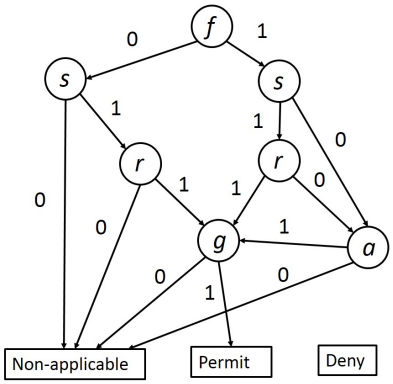


Figure 11 An MTBDD for a simple policy

MTBDDs have three defining characteristics, which are valuable for AC model black box verification. First, they are constructed relative to some fixed ordering on the variables (read from root to terminal): an MTBDD does not allow two subtrees to inspect variables in different orders. This restriction makes MTBDDs a canonical representation up to the chosen variable ordering. Second, MTBDDs maximally share subtrees, in that at most one copy of a subtree appears anywhere in the decision diagram; for example, the three paths to the same grade node in Figure 11. Third, MTBDDs collapse irrelevant variables, meaning that if both values of the same variable refer to the same subtree, the node for that variable is removed, and all references to it are redirected to the shared subtree. The combined effect of the latter two characteristics on the MTBDD in Figure 11 shows the *faculty* and *student* variables are false: rather than enumerate the remaining variables' values, the MTBDD refers directly to the *not-applicable* terminal.

While in the worst case the number of nodes in an MTBDD is exponential in the number of variables, in practice the number of nodes is often polynomial or even linear. This representation therefore gains dramatically over naive DNF in practice. Operations that combine MTBDDs tend to be efficient in practice; most policy verification needs yield MTBDDs with at most the product of the numbers of nodes of the originals.

Developed in Racket (formal PLT) scheme, Margrave [MARGRAVE] is a software tool suite written in PLT Scheme 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. The variables for the policy shown in Figure 11 would correspond to *role*=faculty, *role=student*, *action=receive*, *action=assign,* and finally *resource=grade* (the association between attributes, such as *role*, with values, such as *faculty*, is stated 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. For example, the left two MTBDD models in Figure 12, the MTBDD models for rules saying that *faculty* attempting to *assign grades* should yield *permit* and *students* attempting to *assign grades* should yield *deny* appear. The third MTBDD model represents the result of combining these two rules using *rule-combining* algorithms.

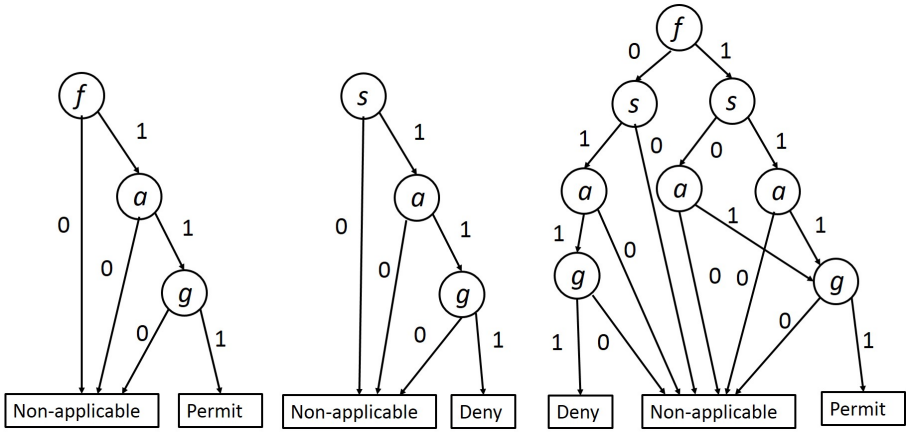


Figure 12 Sample MTBDDs for rules and rule combination.

Margrave allows the user to specify various forms of safety requirements in the Scheme programming language. Margrave's API can verify these safety requirements and if there exist any counterexamples (being specific requests) that violate the specified safety requirements, these counterexamples are produced. Margrave also provides comprehensive change-impact analysis based on the semantic differences between the MTBDD models representing the policies.

The chief novelty of Margrave's current approach lies in embracing the richness of full first-order predicate logic for specifying policies, systems, and queries. Margrave interacts with a user to produce concrete scenarios in response to user queries. Queries cover not only conventional verification-style safety requirements, but also "what if" style questions about the semantic impact of edits. Margrave also allows users to specify policy rules and safety requirement. It then processes user-defined queries about the policy rules and presents scenarios as output. Scenarios are, informally, snapshots of the system—as governed by the policy rules—in which the query holds. [SND13]

Margrave 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.

5.3 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 [AF08] 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 of verifying these two properties. 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 provided a mechanism to detect the inconsistency, incompleteness, applicability and minimality. [AS15] They also mentioned that the MSFOL (many-sorted first order logic) [Gallier87] formula should be converted to a pseudo-Boolean logic formula to analyze it. The proposed tool is a three-step procedure where a 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 [PST96] is based on axiomatic set theory and first order predicate logic, which can be used for describing and modeling AC policies [Hu02]. Z notation using set theory forms an adequate basis for building the AC model, which allows 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 model is meaningful.

5.4 Mutation Testing

Black-box test methods (as described in Section 5.1 through 5.3) may not be sufficient to guard against some unexpected behaviors that are embedded in the AC model as demonstrated in 4.2. To address that, the mutation test of the white box test method generates additional test inputs to cover policy-related entities that are not covered by the black box test methods.

The following illustrates mutation verification of safety requirement through a simple example. This ABAC policy formalizes a university's AC model on assigning and accessing grades. It has two subjects: *faculty* and *student*, two resources: *internal_grades* and *external_grades*, and three actions: *assign*, *view*, and *receive* [Fisler et al]. For this example, we expect the following safety requirements to hold:

S_1 -- There do not exist members of *student* who can *assign external_grades*.

S_2 -- All members of *faculty* can *assign* both *internal_grades and external_grades*.

S_3 -- There exists no combination of subjects such that a user with those subjects can both *receive* and *assign* the resource *external_grades*.

Safety requirement S_1 is intuitive since we certainly do not want students to assign grades. Safety requirement S_2 is to ensure that indeed faculty members can assign grades. S_3 is an example of separation-of-duty since we do not want anyone to assign their own grade, an apparent conflict of interest.

Figure 13 shows the example AC policy model. To keep the example readable and concise, the model is written as simple *If Then* statements, which do not immediately satisfy these three safety requirements. It is essentially a separation-of-duty (SOD) constraint that restricts any request from having both the *faculty* and *student* subjects. All three safety requirements hold in the model.

1 If **subject** = *faculty*
2 and **resource** = (*external_grades* or *internal_grades*)
3 and **action** = *(view or assign)*
4 Then
5 **permit**
6 If **subject** = *student*
7 and **resource** = *external_grades*
8 and **action** = *receive*
9 Then
10 **permit**

Figure 13 Rules in an example policy model

The first step of mutation verification is to generate mutant models using a mutation operator that simply inverts each rule's effect by changing **permit** to **deny** or **deny** to **permit** (one at a time for each mutant model). The number of mutant models created by this operator is equal to the number of rules in the model. The example model has only two rules and thus only two mutant models are generated. Figures 14 and 15 show these two mutant models.

1 If **subject** = *faculty*
2 and **resource** = (*external_grades* or *internal_grades*)
3 and **action** = (*view* or *assign*)
4 Then
5 **deny**
6 If **subject** = *student*
7 and **resource** = *external_grades*
8 and **action** = *receive*
9 Then
10 **permit**

Figure 14 The first mutant model

1 If **subject** = *faculty*
2 and **resource** = (*external_grades* or *internal_grades*)
3 and **action** = (*view* or *assign*)
4 Then
5 **permit**
6 If **subject** = *student*
7 and **resource** = *external_grades*
8 and **action** = *receive*
9 Then
10 **deny**

Figure 15 The second mutant model.

The second step of mutation verification is to determine which safety requirements hold for the original model and each mutant model. The mutant is said to be killed by safety requirement if the safety requirement holds for the original model but does not hold for the mutant model. In other words, the safety requirement reveals the fault seeded in the mutant model. The greater the number of mutants killed, the more comprehensive the original model is in covering the safety requirements.

The first mutant model in Figure 14 does not satisfy S_2 and thus the first mutant is killed. Recall S_2 seeks to ensure the policy that all faculty members can assign grades. Since the *fault* in Figure 14 is precisely the rule that grants this access, the safety requirement is apparently violated. Figure 16 illustrates the output from the black box model verification using Margrave (Section 5.2) on the first mutant model. Each counterexample (i.e., request) is represented as a bit mask where each bit corresponds to the specific attribute-id on Lines 2–11. If the bit is 0, then the corresponding attribute value is not present whereas if the bit is 1 then the corresponding attribute value is present. As expected, the given concrete counterexamples are for a *faculty* to *assign internal_grades* and for a *faculty* to *assign external_grades*. These two counterexamples correspond to Lines 15 and 16 respectively. Access is denied for both requests, indicating a violation of safety requirement S_2.

```
1 Counterexample:
2 1:/Action, command, receive/
3 2:/Action, command, OTHER/
4 3:/Resource, resource-class, external_grades/
5 4:/Resource, resource-class, OTHER/
6 5:/Subject, subject, student/
7 6:/Subject, subject, OTHER/
8 7:/Action, command, view/
9 8:/Action, command, assign/
10 9:/Resource, resource-class, internal_grades/
11 10:/Subject, subject, faculty/
12 1
13 1234567890
14 {
15 00000-0111
16 00100-0101
17 }
```

Figure 16 Counterexamples for the mutant model in Figure 13.

The second mutant model in Figure 15 is not killed by any of the three safety requirements, reflecting that the original model is not comprehensive and does not completely "cover" the safety requirements.

Mutation verification serves the safety requirement with the two purposes: (1) to quantify how thoroughly the model covers safety requirements (2) to facilitate model changes such that the model covers all safety requirements or vice versa.

In some cases, instead fixing a model, the mutation verification serves the purpose to patch the safety requirements, thus, fixing the policy. For example, adding safety requirement S_4 as below fails to hold for the second mutant policy in xx4.

S_4 All members of *student* can *receive external_grades*.

In Summary, Figure 17 illustrates the necessary inputs and resultant outputs of the mutation verification. The inputs are the model under test and, in this case, a single mutation operator. The mutator then generates a set of mutant models, each with a single fault. The mutation operator generates a mutant for each rule by negating the decision of that rule. Although black box verification executes relatively quickly for the mutants, large models can be used to easily generate thousands of mutant models. An equivalent mutant is a mutant that is syntactically different from the original model while being semantically equivalent. In other words, an equivalent mutant will produce the same result as the original model for all inputs and thus provides no benefit and result in an artificial lowering of the mutant killing ratio, giving an under rated and inaccurate quality measurement. We can also determine which safety requirements hold and which do not hold for both the original model and each mutant model. Note that in order to perform verification programmatically, an executable script and Scheme program for the original model and each mutant model may be required [MX07].

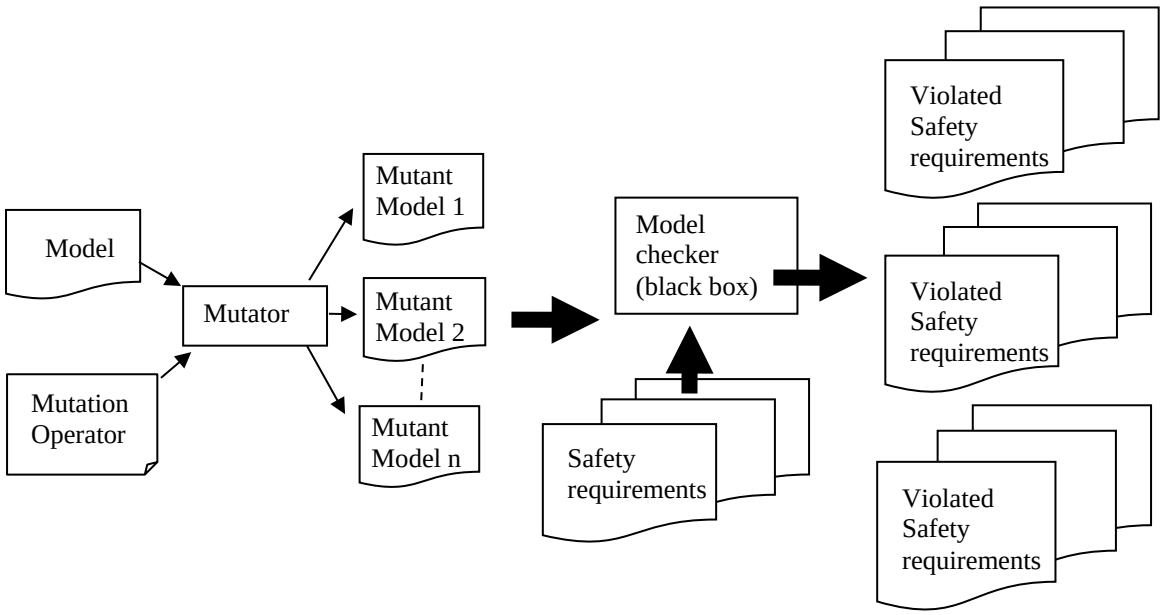


Figure 17 Mutant generation

An important step of mutation testing is to compute the mutant-killing ratio. The mutant-killing ratio is the ratio of the number of mutants killed to the total number of mutants. This ratio serves as a metric to quantify the coverage of safety requirements by the model. A high mutant-killing ratio indicates the original model covers a high number of safety requirements. For example, the mutant coverage (i.e., the mutant-killing ratio) for the safety requirements S_1, S_2, and S_3 in the above example is 50% since only one of two mutants is killed. If S_4 is added, then the mutant-killing ratio is increase to 100%.

The trace files generated by the safety requirement verification described earlier are parsed in order to divide the safety requirement set into four subsets for each mutant. A Venn diagram is illustrated in Figure 18 that describes the relationship of these four sets for a single mutant model. The area inside the box represents the set of all safety requirements. The area inside the left-most circle represents the set of safety requirements that hold true for the original model. Thus the area outside the left-most circle and inside the box is the set of safety requirements that do not hold true for the original model (i.e., these safety requirements fail to be held by the original model). The area inside the right-most circle represents the mutant model that holds false for the safety requirement. Therefore, the area outside the right-most circle and inside the box represents the set of safety requirements that are held true for the mutant model. The area of interest is the intersection of the two circles. If at least one safety requirement is held true for the original model but fails to be held true for the mutant model, then the mutant is killed. If the two circles do not intersect, then the mutant is not killed. A safety requirement that is held true for both the original model and the mutant model has no value in exposing the fault in the mutant model because the safety requirement does not apply to the portion of the model that contains the fault. A safety requirement that is held false for the original model has no value because it is unclear if this false safety

requirement is caused by an error in the model or the safety requirement itself. More specifically, before mutation verification is conducted, these safety requirements must be manually inspected to determine whether they fail due to an error in the model, an error in the safety requirement, or an error in the environment constraints.

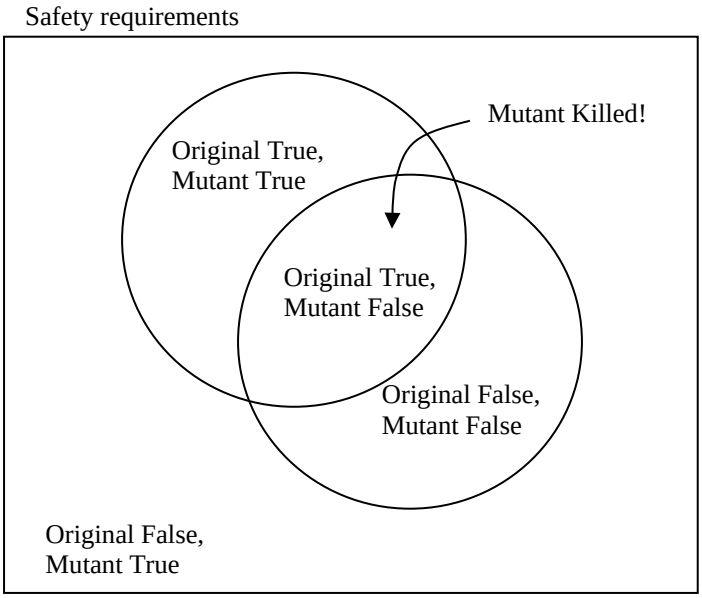


Figure 18 Venn diagram illustrating the four safety requirement states

5.5 Automated Combinatorial Testing

As described in 4.3, a test oracle is required to test the correct implementation of the AC model. However, a real-world AC system is likely to have far too many variables (subjects, actions, objects, conditions, etc.) for generating exhaustive test cases based on the safety requirements. One of the possible solutions for reducing the size therefore efficient in computation for test oracle generation is to apply Automated Combinatorial Testing for Software (ACTS) technology [ACTS] which is a methodology that tests all *t-way* combinations of input parameter values. The most common form is pairwise testing, in which all pairs of input values are covered in at least one test. Higher strength versions of this method cover 3-way, 4-way, or more interactions at least once. The advantage of combinatorial testing for verifying AC implementation is that AC often relies on a small number of discrete values for most parameters. For example, a multi-level security policy (i.e., standard military classification policy) may have levels unclassified, confidential, secret, top-secret, plus a small number of categories, all applied to a collection of resources such as files and programs. For demonstration, a simplified multi-level model, in which each subject (user) has a clearance level *u_l*, and each file has a classification level *f_l*. Levels are given as 0, 1, or 2, which could represent levels such as *Confidential*, *Secret*, and *Top Secret*. A user *u* can read a file *f* if $u\_l \geq f\_l$ (the "no read up" rule), or write to a file if $f\_l \geq u\_l$ (the "no write down" rule) is as the following.

if *u_l* >= *f_l* & act = *rd* then GRANT;
else if *f_l* >= *u_l* & act = *wr* then GRANT;
else DENY;

Figure 19 Simple MLS model

A 2-way covering array for the example is listed in Figure 20. If we had a larger number of parameters, we could produce test configurations that cover all 3-way, 4-way, etc. combinations. (With only three parameters, 3-way interaction is equivalent to exhaustive testing.)

```
Number of parameters: 3
Maximum number of values per
parameter: 3
Number of configurations: 9
------------------------------------
Configuration #1:
1 = u_l=0
2 = f_l=0
3 = act=rd
------------------------------------
Configuration #2:
1 = u_l=0
2 = f_l=1
3 = act=wr
------------------------------------
Configuration #3:
1 = u_l=0
2 = f_l=2
3 = act=rd
------------------------------------
Configuration #4:
1 = u_l=1
2 = f_l=0
3 = act=wr
------------------------------------
Configuration #5:
1 = u_l=1
2 = f_l=1
3 = act=rd
------------------------------------
Configuration #6:
1 = u_l=1
2 = f_l=2
3 = act=wr
------------------------------------
Configuration #7:
1 = u_l=2
2 = f_l=0
3 = act=rd
------------------------------------
Configuration #8:
1 = u_l=2
2 = f_l=1
3 = act=wr
------------------------------------
Configuration #9:
1 = u_l=2
2 = f_l=2
3 = act=wr
```

Figure 20 Covering array for a 2-way example

The next step is to assign values from the covering array to parameters used in the model. For each test, we claim that the expected result will not occur. The model checker determines combinations that would disprove these claims, outputting these as counterexamples. Each counterexample can then be converted to a test with known expected result. As can be seen below, for each of the 9 configurations in the covering array of Figure 20, we create a SPEC claim of the form:

SPEC AG((<covering array values>) -> AX !(access = <result>))

(the SPEC is the syntax of the model checker for the safety requirement specification); This process is repeated for each possible result, in this case either "GRANT" or "DENY", so we have 9 claims for each of the two results as in Figure 21.

Excerpt:

```
...
-- reflection of the assign for access
--SPEC AG ((u_l >= f_l & act = rd ) -> AX (access = GRANT));
--SPEC AG ((f_l >= u_l & act = wr ) -> AX (access = GRANT));
--SPEC AG (!( (u_l >= f_l & act = rd ) | (f_l >= u_l & act = wr ) )
          -> AX (access = DENY));

* SPEC AG((u_l = 0 & f_l = 0 & act = rd) -> AX !(access = GRANT));
* SPEC AG((u_l = 0 & f_l = 1 & act = wr) -> AX !(access = GRANT));
  SPEC AG((u_l = 0 & f_l = 2 & act = rd) -> AX !(access = GRANT));
  SPEC AG((u_l = 1 & f_l = 0 & act = wr) -> AX !(access = GRANT));
* SPEC AG((u_l = 1 & f_l = 1 & act = rd) -> AX !(access = GRANT));
* SPEC AG((u_l = 1 & f_l = 2 & act = wr) -> AX !(access = GRANT));
* SPEC AG((u_l = 2 & f_l = 0 & act = rd) -> AX !(access = GRANT));
  SPEC AG((u_l = 2 & f_l = 1 & act = wr) -> AX !(access = GRANT));
* SPEC AG((u_l = 2 & f_l = 2 & act = rd) -> AX !(access = GRANT));
  SPEC AG((u_l = 0 & f_l = 0 & act = rd) -> AX !(access = DENY));
  SPEC AG((u_l = 0 & f_l = 1 & act = wr) -> AX !(access = DENY));
* SPEC AG((u_l = 0 & f_l = 2 & act = rd) -> AX !(access = DENY));
* SPEC AG((u_l = 1 & f_l = 0 & act = wr) -> AX !(access = DENY));
  SPEC AG((u_l = 1 & f_l = 1 & act = rd) -> AX !(access = DENY));
  SPEC AG((u_l = 1 & f_l = 2 & act = wr) -> AX !(access = DENY));
  SPEC AG((u_l = 2 & f_l = 0 & act = rd) -> AX !(access = DENY));
* SPEC AG((u_l = 2 & f_l = 1 & act = wr) -> AX !(access = DENY));
  SPEC AG((u_l = 2 & f_l = 2 & act = rd) -> AX !(access = DENY));
```

Figure 21 Test cases for test oracle

A model checker produces counterexamples where the input values would disprove the claims specified in Figure 21. Each of these counterexamples is thus a set of test data that would have the expected result of GRANT or DENY (as listed by a * in Figure 21). We now strip out the parameter names and values, from Figure 21 giving tests that can be applied to the system under test. The tests produced are shown below:

u_l = 0 & *f_l* = 0 & act = *rd* -> access = GRANT
u_l = 0 & *f_l* = 1 & act = *wr* -> access = GRANT
u_l = 1 & *f_l* = 1 & act = *rd* -> access = GRANT
u_l = 1 & *f_l* = 2 & act = *wr* -> access = GRANT
u_l = 2 & *f_l* = 0 & act = *rd* -> access = GRANT

u_l = 2 & *f_l* = 2 & act = *rd* -> access = GRANT
u_l = 0 & *f_l* = 2 & act = *rd* -> access = DENY
u_l = 1 & *f_l* = 0 & act = *wr* -> access = DENY
u_l = 2 & *f_l* = 1 & act = *wr* -> access = DENY

These test definitions can now be post-processed using simple scripts to produce a test cases for the test oracle.

5.6 Pseudo-exhaustive Testing

This section describes a method of testing for access control systems (or other systems with formally specified rules) that is *pseudo-exhaustive*, which we define as exhaustive testing of all combinations of attribute values on which an access control decision is dependent [KHFKL16]. An advantage of this method is that it can be used to produce a complete test set, in the sense that all negative cases as well as all positive cases are verified. This approach is analogous to pseudo-exhaustive methods for testing combinational circuits, where the verification problem is reduced by exhaustively testing only the subset of inputs on which an output is dependent, or by partitioning the circuit and exhaustively testing each segment. We can use the basic principle of testing only subsets of attributes on which a decision is dependent, although the partitioning is done in a different manner than for combinational circuits. The structure of the access control problem makes it possible to apply the same principle by rendering the conditions for each *grant* in disjunctive normal form, then considering each term separately.

For example, a rule with attributes *employment_status* and *time_of_day* might be, "If subject is an employee and the hour is between 9 am and 5 pm, then allow entry." The problem with this approach is that n boolean attributes or variables result in potentially 2^n rules. Many such rules may be included in written policy documents, and rules may include a variety of attributes. For any combination of attribute values, the system must implement rules that accurately reflect the written policy. The structure of such rules is typically as follows, where R_i are boolean conditions evaluating the values of one or more attributes:

$R_1 \rightarrow$ *grant*
$R_2 \rightarrow$ *grant*
…
$R_m \rightarrow$ *grant*
else $\rightarrow$ *deny*

which is equivalent to:

$R_1 \rightarrow$ *grant*
$R_2 \rightarrow$ *grant*
…
$R_m \rightarrow$ *grant*
$(\sim R_1)\ (\sim R_2)\ldots\ (\sim R_m) \rightarrow$ *deny*

Example: Suppose we have an access rule as shown below:

```
if (a && (c && !d ||e))  grant();
else if (!a && b && !c)  grant();
else deny();
```

This code can be mapped to the following expression:

$(a(c\bar{d}+e) \rightarrow grant)$
$(\bar{a}b\bar{c} \rightarrow grant)$
$((\sim(a(c\bar{d}+e)))(\sim(\bar{a}b\bar{c})) \rightarrow deny)$

The boolean literals may represent conditions, such as *age>18*, or boolean attributes such as *employee*, but the structure will be as shown in the example. That is, a series of expressions specifying subsets of attribute conditions that must be true for access to be granted, followed by a default deny-access rule when none of the attribute expressions have been instantiated to *true*.

Testing an implementation requires showing that the policy specified, *P*, is correctly implemented. The implemented policy *P'* must be shown to produce the same response as *P* for any combination of attributes used as input. That is, for input attributes $x_1,\ldots,x_n$, $P'(x_1,\ldots,x_n) = P(x_1,\ldots,x_n)$.

Confirming that access will be granted for users with the right attributes is easy: we can simply read off the attribute conditions for each *grant* expression and verify that the access control system returns an authorization in each case. The number of such tests is linear in the number of *grant* conditions. However, it is much more difficult to ensure that no invalid combination of attributes will result in authorization. With *n* boolean attributes or variables there are 2^n possible combinations of attributes. For example, it would not be unusual to have 50 conditions or Boolean attributes, resulting in $2^{50} \approx 10^{15}$ combinations, but it must be shown that no combination will improperly allow access.

To make testing tractable, we will use covering arrays of attributes in policies that have been converted to *k*-DNF form. *k*-DNF refers to disjunctive normal form where no term contains more than *k* literals. Recall that a *term* is a conjunction of one or more literals within the disjunction. For example, *abc + de* contains two terms, one with three literals and one with two, so the expression is in 3-DNF form. A 3-way covering will contain all 3-way combinations of variable values (and therefore also all 1-way and 2-way combinations). Where an expression is in *k*-DNF, any term containing *k* literals that is resolved to true will clearly result in the full expression being evaluated to true. For example, a rule in 2-DNF form could be: "`if employee && US_citizen || auditor then grant`". This rule contains one term of two attributes and one term of one attribute, so it is 2-DNF. Because a covering array of strength *k* contains every possible setting of all *k*-tuples and *i*-tuples for $i < k$, it contains every combination of values of any *k* literals.

Covering array generation tools, such as ACTS, make it possible to include constraints that prevent the inclusion of variable combinations that meet criteria specified in a first order logic style syntax. For example, if we are testing applications that run on various combinations of operating systems and browsers, we may include a constraint such as `'OS = "Linux" => browser != "IE"'`. Constraints are typically used in situations such as this, where certain combinations do not occur in practice, and therefore should not be included in tests.

Method: Let *R* = rule antecedents (left hand side of an implication rule such as *p* in $p \rightarrow q$) of one or more policy rules being tested in *k*-DNF, and T_i are terms (conjuncts of one or more attributes) in *R*. For the example included in the introduction, terms T_i of *R* would be $ac\bar{d}$, *ae*, and $\bar{a}b\bar{c}$. *R* is not necessarily the complete policy; it may be the set of rules associated with a particular resource that we wish to test, for example.

Positive testing: Generate a test set GTEST for which every test should produce a response of *grant*. It must be shown that for all possible inputs, where some combination of *k* input values matches a *grant* condition, a decision of *grant* is returned. Construct test set GTEST with one test for each term of *R* as follows:

$$\text{GTEST}_i = T_i \bigwedge_{j \neq i} \sim T_j$$

The construction ensures that each term in *P* is verified to independently produce a response of *grant*. Negating each term T_j, $i \neq j$, prevents masking of a fault in the presence of other combinations that would return the same result. For example, if a rule condition is $ab + cd \rightarrow grant$, inputs of 1100, 1101, 1110 could be used for testing $ab \rightarrow grant$. However, input 1111 would not detect the fault if the system ignores variable *a* or *b*, because the condition *cd* would cause a *grant* decision, and no other *grant* predicates would be evaluated. One such test is required for each term in a *grant* rule, so for *m* rules with an average of *p* terms each, the number of tests required is proportional to *mp*.

Negative testing: Generate a test set DTEST for which every test should produce a response of *deny*. It must be shown that for all possible inputs, where no combination of *k* input values matches a *grant* condition, a decision of *deny* is returned.

> DTEST = covering array of strength *k*, for the set of attributes included in *R*, with constraints specified by ~*R*.

Note that the structure of the access control rule evaluation makes it possible to use a covering array for DTEST, compressing a large number of test conditions into a few tests. Because a *deny* is issued only after all *grant* conditions have been evaluated, masking of one combination by another can only occur for DTEST when a test produces a response of *grant*. In such a case, an error has been discovered, which can be repaired before running the test set again. Since DTEST is a covering array, the number of tests will be proportional to $v^k \log n$, for *v* values per attribute (normally *v*=2 since most will be boolean conditions), and *n* attributes. For *m* rules, the number of tests is multiplied by the constant *m*.

Example: Table 1 gives a set of boolean attributes *a* through *e*, where each row defines values for the attributes that determine a decision, either *grant* or *deny*. Thus a covering array for the antecedent *R* of a rule in 3-DNF such as ($ac\bar{d} + \bar{a}b\bar{c} \rightarrow grant$) is given in Table 1. The total number of 3-way combinations covered is the number of settings of three binary variables multiplied by the number of ways of choosing three variables from five, i.e., $2^3 \binom{5}{3} = 80$.

Table 2 shows a covering array for this set of variables generated using ~ *R* as a constraint. That is, the two terms of the rule, $ac\bar{d}$ and $\bar{a}b\bar{c}$, have been excluded from the array, but all other 1-, 2-,

and 3-way combinations can be found in the array. Because $ac\bar{d}$ and $\bar{a}b\bar{c}$ are the only conditions under which access should be granted, the array in Table 2 should result in a *deny* response from the access control system for every test. Collectively, the tests include all 78 3-way settings of attributes that will not instantiate the access control rule to *true*.

Table 1: 3-way covering array

	A	b	c	d	e
1	0	0	0	0	0
2	0	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	0	0	1	1
6	1	0	1	0	0
7	1	1	0	0	1
8	1	1	1	1	0
9	1	1	0	0	0
10	0	0	1	1	0
11	0	0	0	0	1
12	1	1	1	1	1

Table 2: 3-way covering array with constraint ~*R*

	a	b	c	d	e
1	0	0	0	0	0
2	0	0	1	1	1
3	0	1	1	0	0
4	1	0	0	1	0
5	1	0	1	1	0
6	1	1	0	0	1
7	1	1	1	1	1
8	0	0	1	0	1
9	1	1	0	1	0
10	0	0	0	1	1
11	1	0	0	0	0
12	0	1	1	1	0
13	1	0	0	0	1
14	0	1	1	0	1

Fault detection properties: Collectively, tests from GTEST and DTEST will detect all added, deleted, or altered faults with up to *k* literals. If more than *k* literals are included in the altered term, some faults are still detected. Proofs of these properties can be found in [IWCT 16], which also details characteristics of faults of more than *k* literals which will be detected.

Generalization of method: Although only two responses are considered in the description of the method above, it is simple to generalize to more than two responses. Because multiple responses are possible, instead of Gtest and Dtest for grant and deny results, we will refer to $Ptest_i$ and $Ntest_i$ arrays, for *i* possible results, and $Ptest_{ij}$ and $Ntest_{ij}$ index rows *j* of each array.

- *Positive testing*: Generate a test set $Ptest_i$ for which every test should produce a response output of O_i. It must be shown that for all possible inputs, where some combination of *k*

input values instantiates the condition for O_i to true, a result of O_i is returned. The construction for Gtest is used for each possible response output O_i to produce one positive array for each response O_i.

- *Negative testing*: Similarly, we construct one negative array $Ntest_i$ for each possible result output O_i as a covering array of strength k, for the terms in rules R_i that result in O_i with constraints specified by $\sim R_i$, for each output i.

Note also that the method is not constrained to access control applications, but can be used for any formally specified system where rules can be converted to a k-DNF form.

5.7 Real Time AC Rule Implementation Detection

Most research on AC model or policy verification techniques is focused on one particular model, and almost all of the research is in applied methods, which require the **completed** AC policies as the input for verification or test processes to generate fault reports (as previously mentioned). Even though correct verification is achieved and counterexamples may be generated along with found faults, those methods provide no information about the source of rule faults that might allow conflicts in privilege assignment, leakage of privileges, or conflict of interest permissions. The difficulty in finding the source of fault is increased especially when the AC rules are intricately covering duplicated variables to a degree of complexity. The complexity is due to the fact that a fault might not be caused by one particular rule; for example, rule *x* grants subject/attribute *s* access to object/attribute *o*, and rule *y* denies the group subject/attribute *g*, which *s* is a member of, access to object *o*. Such conflict can only be resolved by removing either rule *x* or *y*, or the *g* membership of *s* from the policy. But removing *x* or *y* affects other rules that depend on them (e.g., a member of subject group *g k* is granted access to object *o*), and removing *s*'s membership in *g* will disable *g*'s legitimate access to other objects/attributes through the membership. Thus, it requires manually analyzing each and every rule in the policy in order to find the correct solution for the fault.

To address the issue, the AC Rule Logic Circuit Simulation (ACRLCS) technique [HS13], which enables the AC authors to detect a fault when the fault-causing AC rule is added to the model, so the fixing can be implemented in real time (on the spot) before adding other rules that further complicate the detecting effort. In other words, instead of checking by retracing the interrelations between rules after the policy is completed, the policy author needs only check the new added rule against previous "correct" ones. In ACRLCS, AC rules are represented in a simulated logic circuit (SLC) (pronounced CELL-see). By simulation, it means ACRLCS is not necessarily implementable by a physical electronic circuit; however, the concept can be implemented and computed through simulated software.

ACRLCS is composed of SLCs representing AC rules specified in Boolean expressions. A SLC should be able to preserve the assignments of AC variables and privilege hierarchies (through inheritance) and evaluate access permission (e.g., grant or deny) from the implemented rule. With this principle, the technique includes two main processes:

- Construct a SLC based on AC variables specified by Boolean expressions, or relation hierarchies specified by relations in an AC rule. In the SLC, each AC variable is represented by an input switch, and the rule logic operator and hierarchy relation are simulated by logic gates.

- Develop an efficient algorithm to detect rule faults in the policy by triggering input switches representing AC rule variables under verification in the currently constructed SLC. Rule faults are generated as positive signal outputs from the SLC, indicating conflicts in privilege assignments caused by a new added rule.

Figure 22 implements a simple Bell-Lapadula [BL73] read property of MLS policy model, under which *Top_Secret* rank users can read objects in *Top_Secret*, *Secret*, and *Confidential* ranks; *Secret* rank users can read objects in *Secret* and *Confidential* ranks; and *Confidential* rank users can only read objects in the *Confidential* rank.

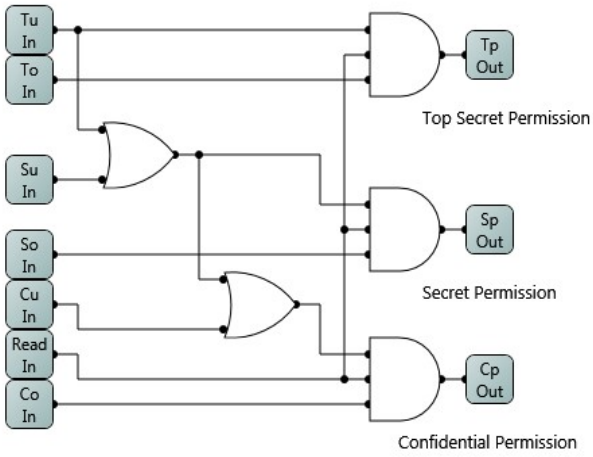


Figure 22 MLS policy model

5.8 Test scheme

General AC system testing framework (as in Figure 23) contains four major functions using the methods as stated in Section 4. The AC Rule Real-time Error detector is used optionally for designing the initial AC models [HS13]. 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 are 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 (as described in Section 3.1, 4.1 and 5.1). 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 sent to White Box Tester for coverage and confinement check (as described in Section 3.2, 4.2 and 5.2). 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, proceed 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 the test oracle (as described in Section 4.3 and 5.3).

Note that the components in Figure 23 and steps listed above 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 Figure 23 except that the Black Box tested is essential.

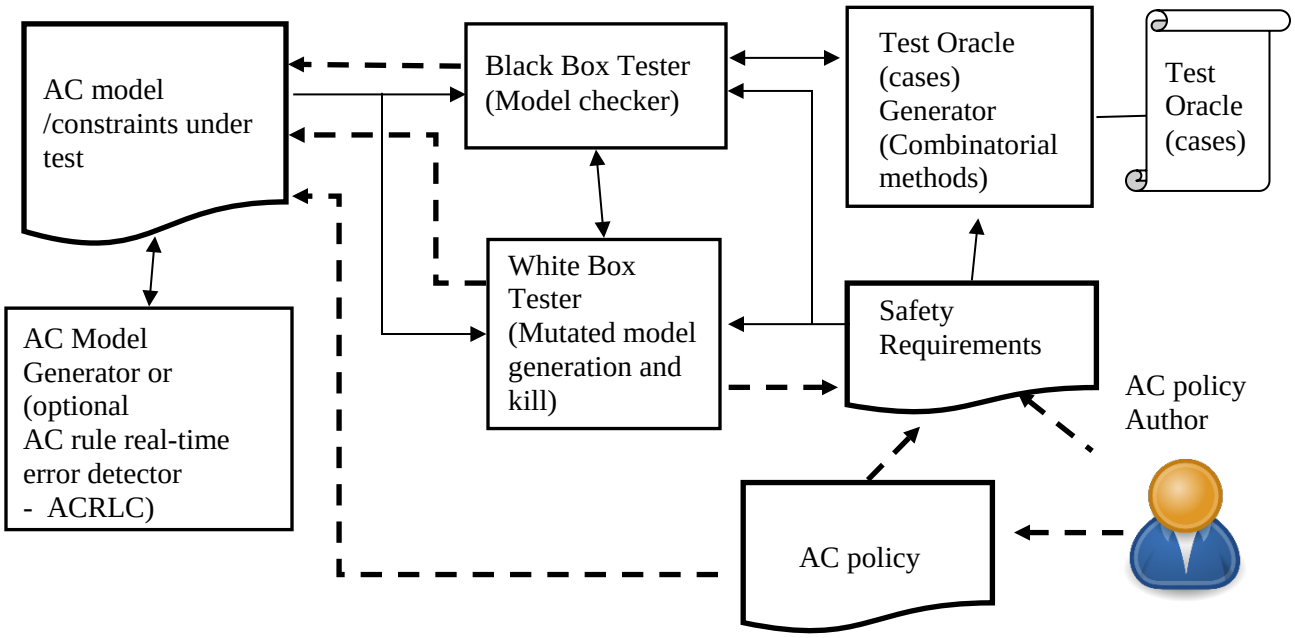


Figure 23 AC testing framework

6 EXAMPLE SYSTEM -- NIST ACPT

NIST's Access Control Policy Tool (ACPT) (Figure 24 from the screen shots) provides (1) GUI templates for composing AC models, (2) safety requirements verification for AC models through an NuSMV (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 as shown in Figure 25 architecture. 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 assures the efficiency of specified AC models, and eliminates the possibility of making faulty AC models that leak the information or prohibit legitimate information sharing.

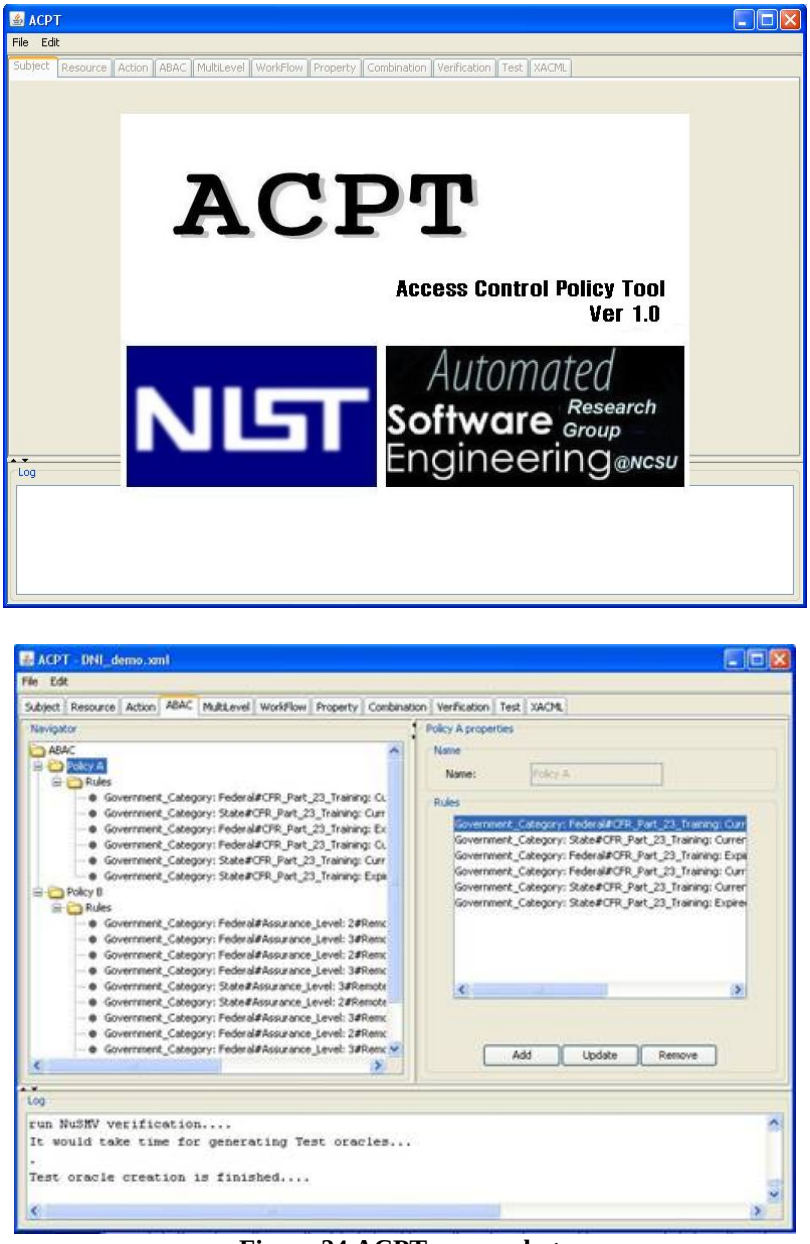


Figure 24 ACPT screen shots

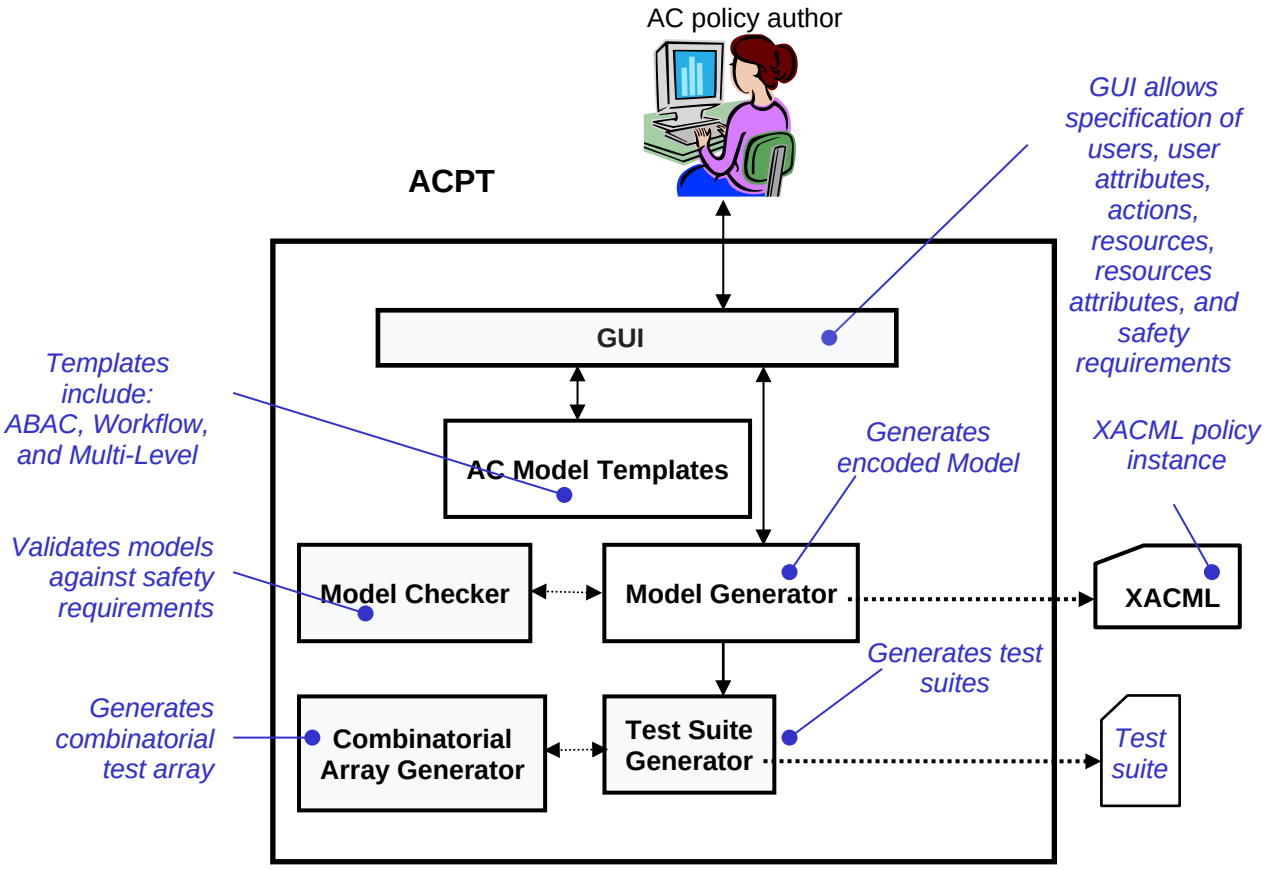


Figure 25 ACPT architecture

ACPT allows users to specify AC models or their combinations, as well as safety requirements through a 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. Figure 26 shows the function flow of the ACPT, followed by the specification of each function.

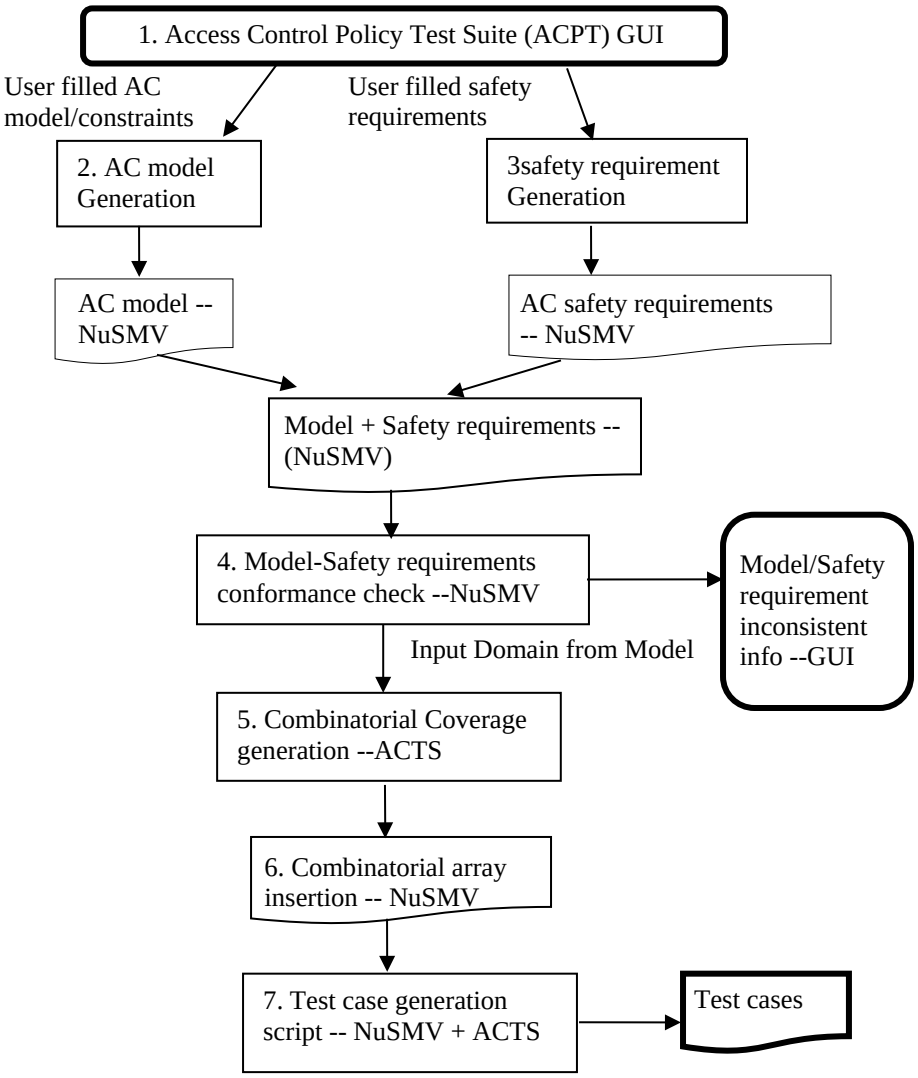


Figure 26 ACPT function flow

1. **ACPT GUI** prompts users to specify required information for the selected AC models from the model templates stored in the database, models such as MLS, RBAC, Work Flow, Chinese, and AC constraints are available for the selection. This interface provides blanks for fill-in (depends on the models selected), and allows users to select more than one model for their specifications. Users are also required to specify AC safety requirements by composing provided predicate and Boolean logic operators with the filled-in AC variables.

2. **AC model generation** produces AC model as part of the final NuSMV program according to the information entered in 1. If more than one model were entered and causing conflicts or errors in the process of model combination, the user will be notified for fixing either their models or safety requirements to resolve the problems.

3. **AC safety requirements generation** produces AC safety requirements as part of the final NuSMV program according to the information entered in 1. The generated safety requirements will be combined with the part of NuSMV model program from 2 to form the final NuSMV program.

4. Model-property conformance check invokes NuSMV Model Checker to output counterexamples found, and then terminate its process if rule conflicts or errors found in the specified AC models (generated by step 2) unless there is no counterexample detected.

5. Combinatorial coverage generation scans the variables in the user specified models, and sends the variables with the ranges of their values to the ACTS combinatorial array generation tool. The outputs from ACTS are then collected for the insertion into the NuSMV program generated in 3.

6. Combinatorial array insertion replaces the variables in the property statement sections of the NuSMV program in 3, the number of safety requirement statements can be as many as the number of array items generated in 5.

7. Test case generation run the resulting NuSMV program from 6. Because of the massive size of the resulting NuSMV program in 6, caused by the size of the variables and their values, support from cluster computing might be required. The counterexamples (if found) from the output of the NuSMV model checker, along with the expected outputs specified in the safety requirement statements of the program will be translated into test cases, which are the final output of the ACPT [HKX08, HXHA10, HXHAS10, HHX12].

Pseudo-exhaustive Testing Function

An implementation of Pseudo-exhaustive testing (Section 5.6) was added to the ACPT. This implementation uses a Boolean based policy as input; the software then ensure that the policy is in *DNF* and then solves for Grant conditions by evaluating the individual portions of the *DNF* policy, and then solves for Deny conditions, by utilizing ACTS with ~R as a constraint.

Example: Given the Boolean Policy of:
`mc && ~oc && ~mr || lo && (mc || cc || oc) || pc`
The explanation of the policy is the following:

Text	Expression
(A) The **{minor consents: mc}** to such health care service; no **{other consent: oc}** to such health care service is required by law, regardless of whether the consent of another person has also been obtained; and the minor has not **{requested that such person: mr}** be treated as the personal representative;	`mc && ~oc && ~mr`
(B) The **{minor may lawfully obtain: lo}** such health care service without the consent of a parent, guardian, or other person acting in loco parentis, and the **{minor: mc}**, a **{court: cc}**, or **{another person: oc}** authorized by law consents to such health care service;	`lo && (mc || cc || oc)`
(C) A **{parent, guardian, or other person acting in loco parentis assents to an agreement of confidentiality: pc}**	`pc`

Figure 27 shows the ABAC Boolean testing interface in ACPT.

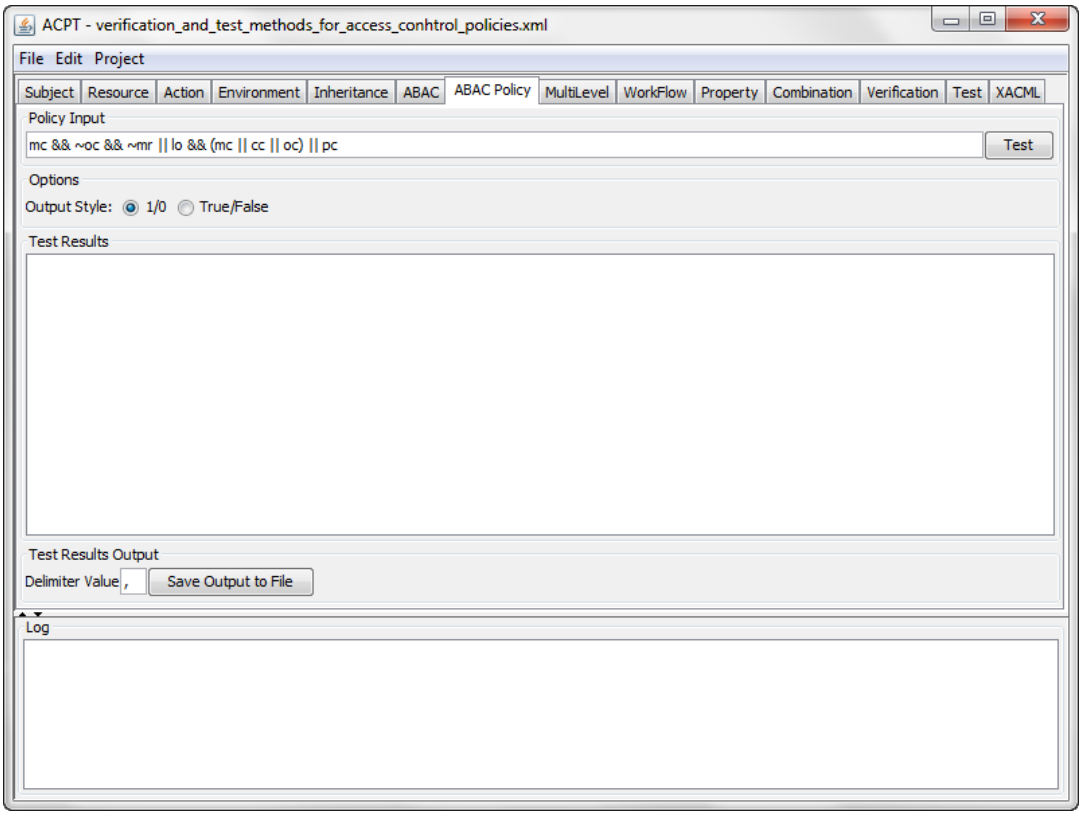


Figure 27 ACPT Showing ABAC Boolean Policy Testing

The expression is then converted within the software into Disjunctive Normal Form (DNF) to be evaluated. The DNF form of the expression can then be split into sections on each `OR` operator. Each section can be evaluated to find a `GRANT` condition (assign `TRUE` for non-negated variables, a `FALSE` for negated variables); for variables not found in that section, a value of `FALSE` is set. Doing this process will generate of all the `GRANT` conditions, as shown in Figure 28.

DNF of the input policy*:*

```
(!mr && !oc && mc) || (cc && lo) || (lo && mc) || (lo && oc) || pc
```

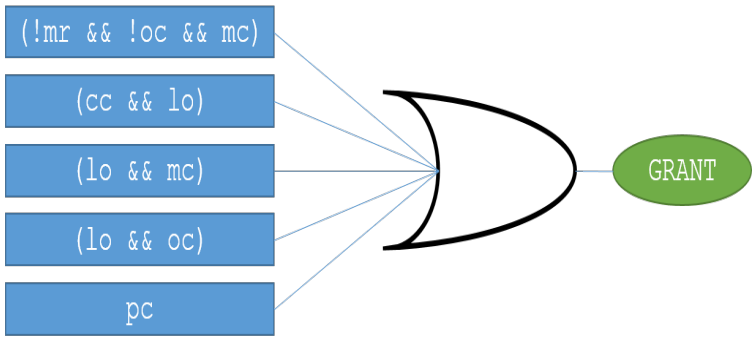


Figure 28 Method for finding Grant Solutions – Each Section through an OR Gate

The resulting GRANT results for the specified policy are below:

Table 3: Result of GRANT Testing

Section of Policy	cc	lo	mc	mr	oc	pc
!mr & !oc & mc	0	0	1	0	0	0
cc & lo	1	1	0	0	0	0
lo & mc	0	1	1	0	0	0
lo & oc	0	1	0	0	1	0
pc	0	0	0	0	0	1

The same table of results is displayed within the software as shown in Figure 29.

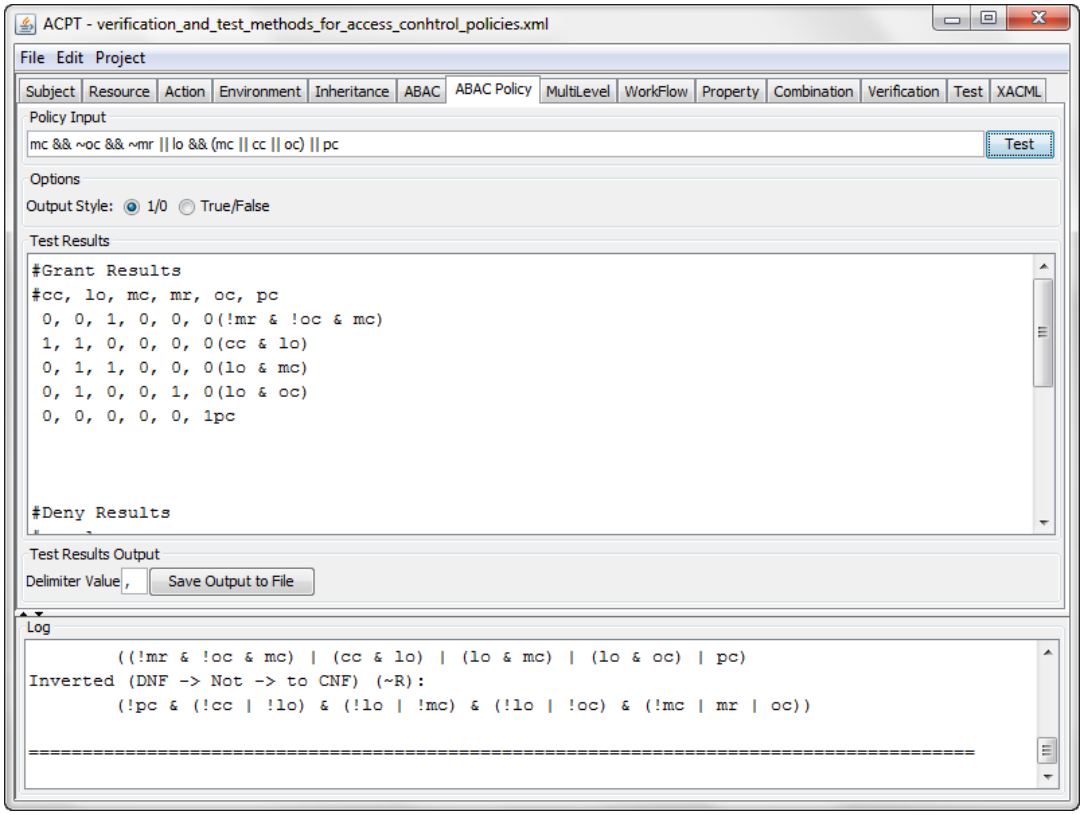


Figure 29 Example Output for Grant Results

To generate the DENY conditions, the DNF of the original policy must be inverted as shown in Figure 30. Then all possible inputs must be found (which, depending on the size of the policy, could take a very long time) – or, using combinatorial testing methods the size of the test set can be reduced; instead of all possible combinations, covering arrays which cover all combinations of inputs can provide coverage. Once the inverted policy is found, it is used as an input into NIST's ACTS software as a constraint to produce the resulting DENY results.

In this case, the inverted *DNF* policy is:

```
(!pc && (!cc || !lo) && (!lo || !mc) && (!lo || !oc) && (!mc || mr || oc))
```

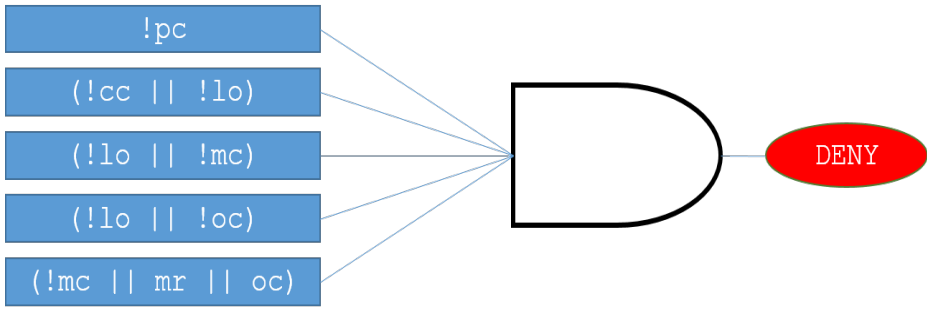


Figure 30 Method for finding Deny Solutions – Each Section through an AND Gate

The resulting deny tests are found below:

Table 4: Result of DENY Testing

cc	lo	mc	mr	oc	pc
1	0	1	1	1	0
1	0	0	0	0	0
0	1	0	1	0	0
0	0	1	0	1	0
0	0	0	1	1	0
0	1	0	0	0	0
0	0	1	1	0	0
1	0	0	1	1	0
1	0	1	0	1	0
1	0	0	0	1	0
1	0	1	1	0	0

The same table of results is displayed within the software, however, the some are not shown in the following screen shot in Figure 31 due to window size.

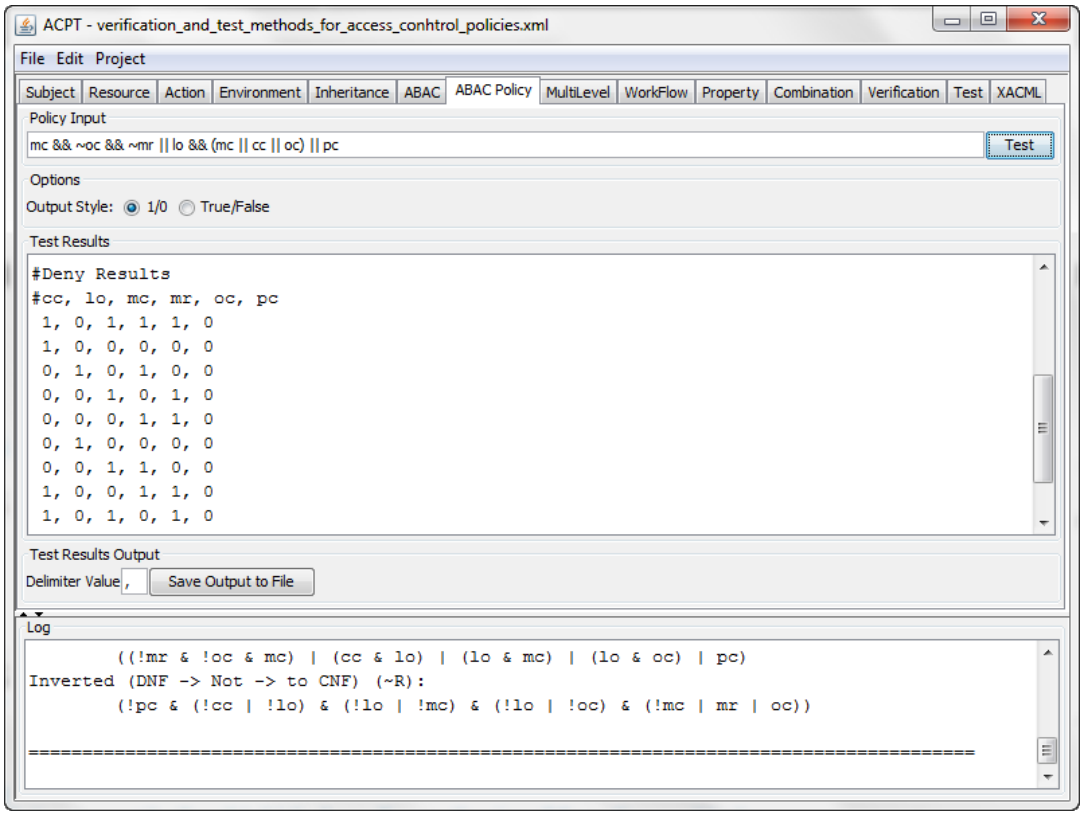


Figure 31 Example Output for Deny Solutions

Relational Testing

In addition to a pure Boolean input policy, some initial support for relational expression testing is included in the software.

Consider the following policy:

```
emp & age > 18; & (fa | emt | med)
```

The relational expression `age > 18;` within the policy can be viewed as a Boolean expression; age has values that will evaluate to true (values greater than 18), and those which will evaluate to false (values less than or equal to 18). The software takes the input policy and replaces the relational expressions with temporary Boolean variables. After the replacement, the software processes the input policy normally. Then the software will solve each relational problem to find values that pass and fail, and updates the results with those values as shown in Figure 32.

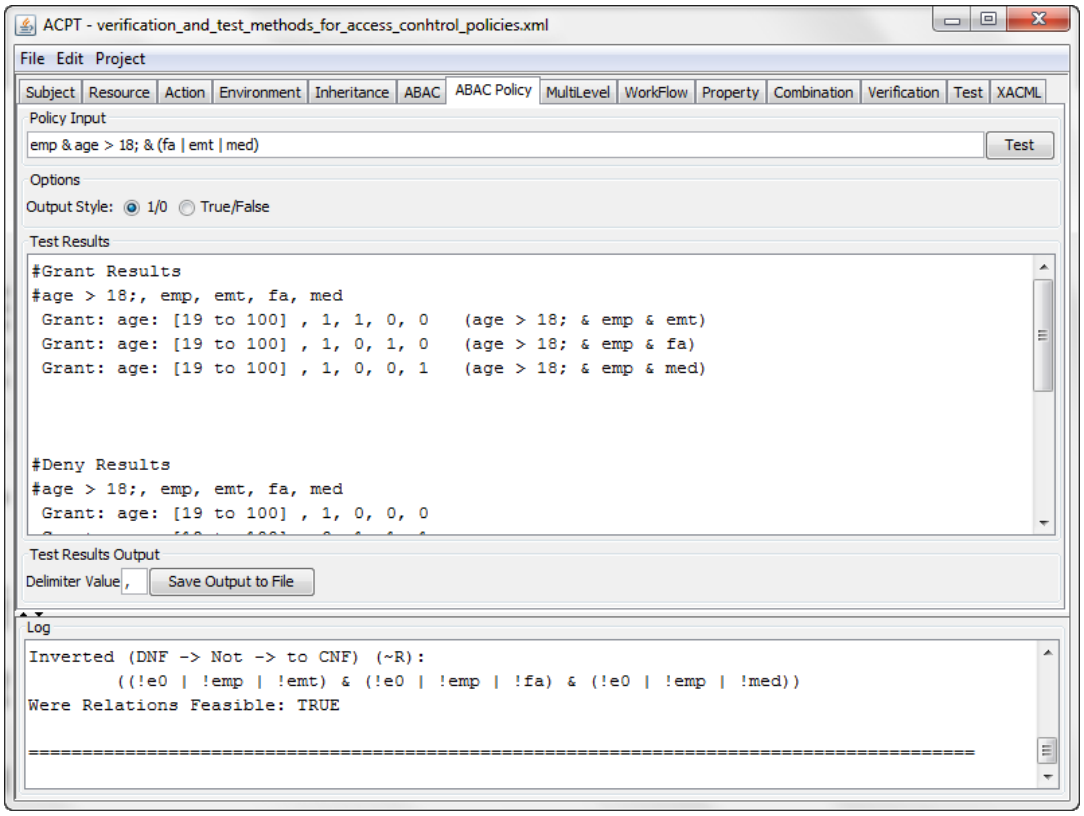


Figure 32 Solving for Relational Expressions

7 CONCLUSIONS

This document 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 access 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 taxonomy of faults that may occur in access control.

To verify safety requirements for AC models, this document provide a general approach that expresses AC models and AC safety requirements in the formal specification of black box models or first order logic checkers for verification. Then black box verifiers verify the specified models against the specified safety requirements. In addition to black box checking, white box checking methods that check the semantic coverage of the safety requirements also conforms to the intentions of the AC model authors. Finally, it is necessary to generate test cases to check the conformance of the implementations to their underlying models.

Conformance verification of access control model and safety requirements 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.

APPENDIX A - GLOSSARY

ABAC

an access control paradigm whereby access rights are granted to users through the use of policies which combine attributes together. The policies can use any type of attributes (user attributes, resource attributes, environment attribute etc.

Access Control

Procedures and controls that limit or detect access to critical information resources. This can be accomplished through software, biometrics devices, or physical access to a controlled space.

Access Control Mechanism

Implementations of formal AC policy such as AC model. Access control mechanisms can be designed to adhere to the properties of the model by machine implementation using protocols, architecture, or formal languages such as program code.

Access Control Model

Formal presentations of the security policies enforced by AC systems, and are useful for proving theoretical limitations of systems. AC models bridge the gap in abstraction between policy and mechanism.

Access Control Policy

High-level requirements that specify how access is managed and who may access information under what circumstances.

ACPT

Access Control Policy Tool.

ACTS

Automated Combinatorial Testing for Software.

Black Box Testing

A method of software testing that examines the functionality of an application without peering into its internal structures or workings. This method of test can be applied to virtually every level of software testing: unit, integration, system and acceptance.

Constraint

Access Control rules or confinements that describe the access privileges of resources for subjects.

Discretionary access control (DAC)

leaves a certain amount of access control to the discretion of the object's owner, or anyone else who is authorized to control the object's access. The owner can determine who should have access rights to an object and what those rights should be.

Extensible Access Control Markup Language (XACML)
A general purpose language for specifying access control policies.

Mandatory access control (MAC)
means that access control policy decisions are made by a central authority, not by the individual owner of an object. User cannot change access rights. An example of MAC occurs in military security, where an individual data owner does not decide who has a top-secret clearance, nor can the owner change the classification of an object from top-secret to secret.

Security Policy
The statement of required protection for the information objects.

Safety Requirements
AC properties, business requirements, specifications of expected/unexpected system security features, or directly translation of policy values. Safety requirements can also include privilege inheritance.

Separation of Duty (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 a paycheck should not also be the one who can prepare them. Separation of duties can be enforced either statically (by defining conflicting roles, i.e., roles which cannot be executed by the same user) or dynamically (by enforcing the control at access time). An example of dynamic separation of duty is the two-person rule. The first user to execute a two-person operation can be any authorized user, whereas the second user can be any authorized user different from the first [SS94]. There are various types of SOD, an important one is history-based SOD that regulate for example, the same subject (role) cannot access the same object for variable number of times.

White Box Testing
(also known as clear box testing, glass box testing, transparent box testing, and structural testing) is a method of testing software that tests internal structures or workings of an application, as opposed to its functionality (i.e. black-box testing).

APPENDIX B – SOD CONSTRAINTS

1. Static Separation of Duties

(Simple) Static Separation of Duties (SSSoD)

A principal may not be a member of any two exclusive roles.

2. Dynamic Separation of Duties

(Simple) Dynamic Separation of Duties (SDSoD)

A principal may be a member of any two exclusive roles but must not activate them at the same time.

Object-based Separation of Duties (ObjSoD)

A principal may be a member of any two exclusive roles and may also activate them at the same time, but he must not act upon the same object through both.

Operational Separation of Duties (OpSoD)

A principal may be a member of some exclusive roles as long as the set of authorizations acquired over these roles does not cover an entire workflow.

History-based Separation of Duties (HistSoD)

A principal may be a member of some exclusive roles and the complete set of authorizations acquired over these roles may cover an entire workflow, but a principal must not use all authorizations on the same object(s). [SLS06]

REFERENCES

[ACTS] Automated Combinatorial testing for Software (ACTS) [Website], National Institute of Standards and Technology. Available at: http://csrc.nist.gov/groups/SNS/acts/index.html.

[AF08] 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. https://doi.org/10.1109/IAS.2008.52.

[AS15] M. Aqib, R. A. Shaikh, "Analysis and Comparison of Access Control Policies Validation Mechanisms," *International Journal of Computer Network and Information Security*, 2015, 1, 54-69. https://doi.org/10.5815/ijcnis.2015.01.08.

[BL73] D.E. Bell, and L.J. LaPadula, *Secure Computer Systems: Mathematical Foundations and Model*, M74-244, MITRE Corp., Bedford, Mass., 1973 (also available as DTIC AS-771543). Available at: http://www-personal.umich.edu/~cja/LPS12b/refs/belllapadula1.pdf.

[CFMYZ93] 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.Available at: http://repository.cmu.edu/cgi/viewcontent.cgi?article=1456&context=compsci.

[Fisler et al 05] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz, "Verification and Change Impact Analysis of Access Control Policies," *Proceeding, 27th International Conference on Software Engineering (ICSE'05)*, pp. 196-205, ACM, New York, NY, 2005. https://doi.org/10.1145/1062455.1062502.

[Gallier87] J. H. Gallier, *Logic for Computer Science: Foundations of Automatic Theorem Proving*, ch. 10, pp. 448–476, Wiley, 1987.

[GMH14] A. Gouglidis, I. Mavridis, V. Hu, "Security policy verification for multi-domains in Cloud systems," *International Journal of Information Security (IJIS13)* 13(2), pp. 97-111, April 2014. https://doi.org/10.1007/s10207-013-0205-x.

[HHX12] J. Hwang, V. Hu, and T. Xie, "Paradigm in Verification of Access Control," Position paper, *Workshop on Metrics and Standards for Software Testing*, NIST Gaithersburg, Maryland, June 20, 2012. https://doi.org/10.1109/SERE-C.2012.14.

[HKX08] V.C. Hu, D.R. Kuhn, and 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. https://doi.org/10.1109/EUC.2008.22.

[HKXH11] 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)* 21(1), February 2011. https://doi.org/10.1142/S021819401100513X.

[Kuhn11] D.R. Kuhn, "Vulnerability hierarchies in access control configurations," *4th Symposium on Configuration Analytics and Automation (SAFECONFIG 2011)*, pp. 1-9, December 2011. https://doi.org/10.1109/SafeConfig.2011.6111679.

[Hu02] V.C. Hu, *The Policy Machine For Universal Access Control*, Dissertation, Computer Science Department, University of Idaho, Idaho, 2002.

[HRU76] M.A. Harrison, W.L. Ruzzo, and J.D. Ullman, "Protection in Operating Systems," *Communications of the ACM* 19(8), pp. 461-471, August 1976. https://doi.org/10.1145/360303.360333.

[HS13] V. Hu and 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., September 8-14, 2013, pp. 494-501. https://doi.org/10.1109/SocialCom.2013.76.

[HXHA10] J. Hwang, T. Xie, V. Hu, and M. Altunay, "Mining Likely Properties of Access Control Policies via Association Rule Mining," *DBSec 2010: Data and Applications Security and Privacy XXIV: 24th Annual IFIP WG 11.3 Working Conference, Rome, Italy, June 21-23, 2010, Proceedings*, Lecture Notes in Computer Science 6166, Springer: Berlin, pp. 193-208. https://doi.org/10.1007/978-3-642-13739-6_13.

[HXHAS10] J. Hwang, T. Xie, V. Hu, and M. Altunay, "ACPT: A Tool for Modeling and Verifying Access Control Policies," *Proceedings of the IEEE International Symposium on Policies for Distributed Systems and Networks (POLICY 2010)*, System Demo, George Mason University, Fairfax, Virginia, USA, July 21-23, 2010, pp. 40-43. https://doi.org/10.1109/POLICY.2010.22.

[MX07] E. Martin and T. Xie, "A Fault Model and Mutation Testing of Access Control Policies," *Proceedings of the 16th International Conference on World Wide Web (WWW 2007)*, Security, Privacy, Reliability, and Ethics Track, Banff, Alberta, Canada, May 2007, pp. 667-676. https://doi.org/10.1145/1242572.1242663.

[KTAK07] S. Kikuchi, S. Tsuchiya, M. Adachi, and T. Katsuyama, "Policy Verification and Validation Framework Based on Model Checking Approach," *Fourth International Conference on Autonomic Computing, 2007 (ICAC '07)*, Jacksonville, Florida, USA, June 11-15, 2007, pp. 1–9. https://doi.org/10.1109/ICAC.2007.31.

[KHFKL16] D.R. Kuhn, V.C. Hu, D.F. Ferraiolo, R.N. Kacker, and L. Yu, "Pseudo-exhaustive Testing of Attribute Based Access Control Rules," *Fifth International Workshop on Combinatorial Testing (IWCT 2016)*, April 13, 2015, Graz, Austria, pp. 51-58. https://doi.org/10.1109/ICSTW.2016.35.

[MARGRAVE] The Margrave Policy Analyzer [Website], available at: http://www.margrave-tool.org/.

[MOHAWK] Mohawk: Automatic Verification of Access-Control Policies [Website], available at: http://kjayaram.mysite.syr.edu/Mohawk.html.

[NIST-IR-7316] V.C. Hu, D.F. Ferraiolo, D.R. Kuhn, *Assessment of Access Control Systems*, NIST Interagency Report 7316, National Institute of Standards and Technology, Gaithersburg, Maryland, September 2006. https://doi.org/10.6028/NIST.IR.7316.

[NIST-IR-7874] V.C. Hu and K. Scarfone, *Guidelines for Access Control System Evaluation Metrics*, NIST Interagency Report 7874, National Institute of Standards and Technology, Gaithersburg, Maryland, September 2012. https://doi.org/10.6028/NIST.IR.7874.

[NIST-SP-162] V.C. Hu, D. Ferraiolo, R. Kuhn, A. Schnitzer, K. Sandlin, R. Miller, and K. Scarfone, *Guide to Attribute Based Access Control (ABAC) Definition and Considerations*, NIST Special Publication 800-162, National Institute of Standards and Technology, Gaithersburg, Maryland, January 2014. https://doi.org/10.6028/NIST.SP.800-162.

[NuSMV] R. Cavada, A. Cimatti, G. Keighren, E. Olivetti, M. Pistore, and M. Roveri, *NuSMV 2.6 Tutorial*, FBK-irst. Available at: http://nusmv.fbk.eu/NuSMV/tutorial/v26/tutorial.pdf.

[NuSMV-M] NuSMV: a new symbolic model checker [Website], available at: http://nusmv.fbk.eu/.

[PST96] B. Potter, J. Sinclair, and D. Till, *An Introduction to Formal Specification and Z*, Second Edition, Prentice Hall: Saddle River, New Jersey, 1996.

[SLS06] A. Schaad, V. Lotz, and K. Sohr, "A Model-checking Approach to Analyzing Organizational Controls in a Loan Origination Process," *Proceedings of the 11th ACM Symposium on Access Control Models and Technologies*, Lake Tahoe, California, June 7-9, 2006, pp. 139-149. https://doi.org/10.1145/1133058.1133079.

[SND13] S. Saghafi, T. Nelson, D. J. Dougherty, "Geometric Logic for Policy Analysis," *International Workshop on Automated Reasoning in Security and Software*

Verification, 2013, pp. 1-9. Available at: http://web.cs.wpi.edu/~tn/publications/snd-arsec13-geometric.pdf.

[SS94] R.S. Sandhu., and P Samarati, "Access Control: Principles and Practice," IEEE Communications Magazine 32(9), September 1994, pp. 40-48. https://doi.org/10.1109/35.312842.

[XACML] XACML resources [Website], OASIS. Available at: http://docs.oasis-open.org/xacml/.