Vulnerability Hierarchies in Access Control Configurations

D. Richard Kuhn
National Institute of Standards and Technology
Gaithersburg, MD 20899
kuhn@nist.gov

Abstract— This paper applies methods for analyzing fault hierarchies to the analysis of relationships among vulnerabilities in misconfigured access control rule structures. Hierarchies have been discovered previously for faults in arbitrary logic formulae [11,10,9,21], such that a test for one class of fault is guaranteed to detect other fault classes subsumed by the one tested, but access control policies reveal more interesting hierarchies. These policies are normally composed of a set of rules of the form "if [conditions] then [decision]", where [conditions] may include one or more terms or relational expressions connected by logic operators, and [decision] is often 2-valued ("grant" or "deny"), but may be *n*-valued. Rule sets configured for access control policies, while complex, often have regular structures or patterns that make it possible to identify generic vulnerability hierarchies for various rule structures such that an exploit for one class of configuration error is guaranteed to succeed for others downstream in the hierarchy.

A taxonomy of rule structures is introduced and detection conditions computed for nine classes of vulnerability: added term, deleted term, replaced term, stuck-at-true condition, stuck-at-false condition, negated condition, deleted rule, replaced decision, negated decision. For each configuration rule structure, detection conditions were analyzed for the existence of logical implication relations between detection conditions. It is shown that hierarchies of detection conditions exist, and that hierarchies vary among rule structures in the taxonomy. Using these results, tests may be designed to detect configuration errors, and resulting vulnerabilities, using fewer tests than would be required without knowledge of the hierarchical relationship among common errors. In addition to practical applications, these results may help to improve the understanding of access control policy configurations.

Keywords- access control; change impact analysis; configuration analysis;

I. INTRODUCTION

Access control is one of the central problems in computer security, and many access control models have been defined, including discretionary access control (DAC), mandatory access control (MAC), and role based access control (RBAC) among the most commonly used types [2], and less standardized forms of access control are often used in network appliances such as firewalls. Access control policies often become large and complex, and rule configurations evolve over time as functions are added or changed, or additional systems are connected. It is estimated that configuration errors account for up to 80% of network vulnerabilities [4]. For defensive purposes, policies must be tested to ensure that they behave as expected, and because policies may include hundreds or even thousands of rules [13], a large number of tests may be needed.

In this paper we show that there is a hierarchical relationship among vulnerabilities in access control systems, 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 paper demonstrates the existence of these hierarchies for a variety of access control rule configurations, and shows how the results may be used to reduce the number of tests required.

The analysis of hierarchies of vulnerabilities can be compared with similar analyses for testing [11,10,9,21], where a vulnerability corresponds to a fault and a test corresponds to an exploit for that vulnerability. However, for testing this analysis is normally applied to arbitrary logic formulae while access control rules are typically implemented in common patterns such as "*if A then Grant; else if B then Grant; else … ; else Deny*". Two types of hierarchies can be shown: those specific to a particular access control policy, and generic forms that are determined by the structure of access control rules. A taxonomy of rule structures is defined and for various possible flaws (e.g., deleted term, added term), conditions under which these faults can be detected are shown to form mathematical structures in which detection conditions for some vulnerabilities subsume those for others.

A *vulnerability* is defined in the RFC 2828 [15] as *"A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy"*. To formalize this definition, we distinguish the specified policy as P and the policy as configured by coded rules on an operational system as R. That is, applications that are implemented must conform to the policy, P, but due to human error or system failures, the policy as implemented may not be correct with respect to the defined policy P. The policy as implemented by administrators using access control rules is designated R. If $R = P$, then the policy has been implemented correctly and by definition there are no vulnerabilities with respect to this policy (although there may be problems unanticipated by policy designers that are later considered vulnerabilities). If $R \neq P$, then one or more vulnerabilities exist with respect to the policy such that unauthorized access or denial of service may be allowed.

Access control policies often take the form of a sequence of rules composed of conditions that result in a decision to grant or deny access. For example, where the C_i contain one or more conditional expressions:

```
if (C1) then grant;
if (C2) then deny;
```

```
        if ($C_3$) then grant;
        deny;  /* default */
```

Conditions in turn include references to the components of access control policies: e.g., subjects, resources, groups, roles, attributes, permissions, and access requests. For example:

```
        if ((role = teller OR role = supervisor)
        AND(request  =  account_balance))  then
        grant;
```

Access control policies may be configured on running systems with dozens or hundreds of rules, sometimes interacting in ways that are determined dynamically depending on system events. This level of complexity requires careful evaluation to ensure that rules are implemented and modified correctly, and a variety of systems have been developed to analyze and test access control policies [5,6,14]. To enable this type of analysis, policies must be represented formally. In this paper, policies are represented and analyzed using mathematical logic formulae that could be mapped to representations for tools such as Margrave [5] and ACPT [6]. For example,

```
        if (c1) then grant;
        else if (c2) then grant;
```

may be represented as $(c_1 \rightarrow grant) \cdot (c_2 \rightarrow grant)$. At first glance, it may seem that the policy above could be represented with "OR" operators between the clauses, as $(c_1 \rightarrow grant) + (c_2 \rightarrow grant)$. But note that the second expression is equivalent to $(c_1 c_2 \rightarrow grant)$, i.e., grant only if both c_1 and c_2 are true, which is clearly not what is intended and not the way that the policy would be processed with the code above. The first expression simplifies to $(c_1 + c_2 \rightarrow grant)$, which is the appropriate meaning and consistent with the code.

While many applications have hard-coded access rules, frameworks capable of implementing access control have become available. XACML [1,16] is one of the better known examples. XACML supports an approach often described as "attribute-based" access control, where XACML attributes can be subjects, actions, or resources. An attribute-ID, such as "clearance level" identifies the attribute, and may take on different attribute values, such as Secret or TopSecret. Implementer-defined rules are used to determine the decision for access requests that present attributes to be processed by the rule engine. XACML decisions can be permit, deny, or not applicable, if there is no rule that matches the attributes in the request. The option of a "not applicable" decision is needed for the generic framework approach more so than hard coded application specific rules because the framework must be able to accommodate changing, arbitrarily complex rule sets. An analysis of vulnerabilities in access rule structures must therefore consider more than the conventional binary decisions of access control.

The organization and main results of this paper are as follows: Section II introduces a taxonomy of access control rule structures; Sect. III explains the computation of vulnerability conditions; Sect. IV describes a variety of flaws in access control rules that result in vulnerabilities; and Sect. V demonstrates how these flaws result in vulnerability hierarchies for the various rule structures in the taxonomy. We conclude with a discussion of the implications of these results for testing access control configurations, and other application to analyzing configuration changes.

II. Taxonomy of Rule Structures

An access control *policy*, *P*, is implemented by a set of *rules*, *R*. A *vulnerability* will be defined here as a condition under which the decision from rules *R* differs from the intended result specified by policy *P*, that is, where $R \neq P$. Vulnerabilities may be 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:

$R_c \Rightarrow P_c$: a (possibly partial) denial of service (because $R_c \subset P_c$).

$R_c \Leftarrow P_c$: unauthorized access (because $R_c \supset P_c$)

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:

$R_c \Rightarrow P_c$: unauthorized access (because $R_c \subset P_c$)

$R_c \Leftarrow P_c$: a (possibly partial) denial of service (because $R_c \supset P_c$).

otherwise: possible combination of unauthorized access or denial of service

Example: Suppose a policy specifies that access is to be granted only when [*the subject is an employee*], designated as *e*, and either [*the time is during working hours*], *h*, or [*the subject is a supervisor*], *s*, and that the policy is implemented as "*if e&h → grant; if e&s → grant; deny;*". The Grant condition is specified as $e \cdot h + e \cdot s$. If the policy is implemented incorrectly so that the second rule is not included, then the Grant condition is $e \cdot h$. Thus $e \cdot h \Rightarrow e \cdot h + e \cdot s$ and there is a denial of service where $e \cdot \overline{h} \cdot s$. (This paper follows conventional practice in using · (or juxtaposition) for boolean *and*, + for *or*, with $\oplus$ signifying *exclusive-or*).

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 (e.g., A *AND* B *AND* C; A *OR*

B *OR* C). 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. Clearly the set of policies thus defined does not cover the universe of possible structures, but a large number of practical policy rule structures can be captured in this manner. Note also that the access control rule structures defined here are not the same as access control policy models. A policy model ensures specific properties are maintained among the elements of the model. For example, the multilevel security policy model guarantees that a user cleared only to Secret cannot read data labeled TopSecret. One or more rule structures could be used to implement a particular policy model.

Here we define the following rule structure attributes and possible 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.

```
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;
```

This structure is modeled by:

$$(\ (c_{11} \cdot \ldots \cdot c_{1n1}) \rightarrow \text{grant})$$
$$\cdot \ ((c_{21} \cdot \ldots \cdot c_{2n2}) \rightarrow \text{grant})$$
$$\ldots$$
$$\cdot \ ((c_{k1} \cdot \ldots \cdot c_{kn3}) \rightarrow \text{grant})$$
$$\cdot \ (\sim(c_{11} \cdot \ldots \cdot c_{1n1}) \cdot \sim(c_{21} \cdot \ldots \cdot c_{2n2}) \ldots \cdot \sim(c_{k1} \cdot \ldots \cdot c_{kn3}) \rightarrow \text{deny})$$

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.

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

This structure is modeled by:

$$(\ (c_{11} + \ldots + c_{1n1}) \rightarrow \text{grant})$$
$$\cdot \ ((c_{21} + \ldots + c_{2n2}) \rightarrow \text{grant})$$
$$\ldots$$
$$\cdot \ ((c_{k1} + \ldots + c_{kn3}) \rightarrow \text{grant})$$
$$\cdot \ (\sim(c_{11} + \ldots + c_{1n1}) \cdot \sim(c_{21} + \ldots + c_{2n2}) \ldots \cdot \sim(c_{k1} + \ldots + c_{kn3}) \rightarrow \text{deny})$$

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 [16] standard is one such widely used framework for implementing access control policies. An XACML rule set includes clauses for subjects, resources, and actions, where rules include matching conditions for these three attributes.

For example, an XACML rule may specify "if (role = engineer OR role = technician) AND (database = test_results) AND (action = append) then GRANT". (XACML uses an XML syntax, but for readability, this example is given in natural language.) Rules may have decisions of grant, deny, or no match in cases where none of the rule predicates match the set of attributes presented to the XACML decision system. In this paper, we consider only XACML rules where a series of grant rules are offered, with a default deny. In the syntax below, we use *s*, *r*, and *a,* for subject, resource, and action terms, but the extension to other CNF rule sets is obvious.

```
if ($s_{11} + \ldots + s_{1n1}$) $\cdot$ ($r_{11} + \ldots + r_{1n1}$) $\cdot$ ($a_{11} + \ldots + a_{1n1}$)
then grant;
```

```
if $(s_{21} + ... + s_{2n1}) \cdot (r_{21} + ... + r_{1n1}) \cdot (a_{21} + ... + a_{2n1})$
then grant;
. . .
if $(s_{k1} + ... + s_{kn1}) \cdot (r_{k1} + ... + r_{kn1}) \cdot (a_{k1} + ... + a_{kn1})$
then grant;
else deny;
```

This structure is modeled by:

$((s_{11} + ... + s_{1n1}) \cdot (r_{11} + ... + r_{1n1}) \cdot (a_{11} + ... + a_{1n1})$
$\rightarrow$ grant)
$\cdot ((s_{21} + ... + s_{2n1}) \cdot (r_{21} + ... + r_{1n1}) \cdot (a_{21} + ... + a_{2n1})$
$\rightarrow$ grant)
. . .
$\cdot ((s_{k1} + ... + s_{kn1}) \cdot (r_{k1} + ... + r_{kn1}) \cdot (a_{k1} + ... + a_{kn1} \rightarrow$
grant)
$\cdot (\sim((s_{11} + ... + s_{1n1}) \cdot (r_{11} + ... + r_{1n1}) \cdot (a_{11} + ... + a_{1n1})$
$... \cdot \sim((s_{k1} + ... + s_{kn1}) \cdot (r_{k1} + ... + r_{kn1}) \cdot (a_{k1} + ... +$
$a_{kn1}) \rightarrow$ deny)

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 are replaced by *n* possible decisions other than *deny*.

III. Vulnerability Detection Conditions

As defined above, a *vulnerability* is a faulty implementation of an access control policy. The detection conditions for a flaw in a policy *P* are given by the boolean difference, $P \oplus P'$, where *P'* represents formula *P* with the fault inserted. If *P'* is *P* with Fault 1, and *P''* is *P* with Fault 2, and $P \oplus P' \Rightarrow P \oplus P''$, then a test that detects Fault 1 will also detect Fault 2 [11]. In this case we say that Fault 1 subsumes Fault 2. Consequently, a test constructed to detect Fault 1 will detect Fault 2 as well, eliminating the need to construct tests specifically for Fault 2. Depending on the form of boolean expressions in the specification, and the fault classes defined, a hierarchy may be established such that one or more root nodes subsume other fault classes [9, 10, 11,12, 21]. Then tests defined for the root fault class(es) will detect the other classes subsumed by these, obviating the need to develop additional tests.

Example1: Consider the policy from the previous example which specifies that access is to be granted only when [*the subject is an employee*], designated as *e*, and either [*the time is during working hours*], *h*, or [*the subject is a supervisor*], *s*, a policy specified as $P = (eh + es \rightarrow G)(\sim(eh + es) \rightarrow \overline{G})$. If rule *R* is implemented incorrectly as $(eh \rightarrow G)(\sim(eh) \rightarrow \overline{G})$, then $eh \Rightarrow eh + es)$ and there is a denial of service where $e \cdot \overline{h} \cdot s$. To detect this vulnerability, compute

$$(eh + es) \rightarrow G)(\sim(eh + es)) \rightarrow \overline{G}) \oplus (eh \rightarrow G)(\sim(eh) \rightarrow \overline{G})$$
$$= e\overline{h}s$$

That is, a result that differs from the correct policy occurs where the subject is an employee and supervisor and the time is out of hours. A test input of $e\overline{h}s$ will detect that the implemented policy does not behave according to the specified policy because the subject is improperly denied access.

A different situation occurs if the implemented policy allows access that is not authorized by the specified policy. For example if the implemented policy fails to check hours, i.e., if $(e + es \rightarrow G)(\sim(e + es) \rightarrow \overline{G})$ is implemented, then unauthorized access will be allowed when the subject is an employee but not a supervisor and time is out of working hours:

$$(eh + es \rightarrow G)(\sim(eh + es) \rightarrow \overline{G})$$
$$\oplus (e + es \rightarrow G)(\sim(e + es) \rightarrow \overline{G}) = e\overline{h}\,\overline{s}$$

This type of analysis can be applied to more complex rules.

Example 2: A 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:

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

That is, for inputs of either $\overline{a}\ b\,c\,\overline{d}$ or $\overline{a}\ b\,c\,\overline{e}$, 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

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

Thus a test with $b\ c\ \overline{d} + b\ c\ \overline{e}$ (and either *a* or $\overline{a}$) would detect the error. Since $P \oplus P_{tc} = \overline{a}\,(P \oplus P_{nc})$, it is easy to see 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 $\bar{G}$.

Extreme cases may also be analyzed in this manner: a faulty policy that should grant access under condition p but grants access in *all* cases has vulnerability condition $\bar{p}$: $(p \rightarrow G)(p \rightarrow \bar{G}) \oplus (1 \rightarrow G) = \bar{p}$. A faulty policy with the opposite result, that always denies access, has vulnerability condition p: $(p \rightarrow G)(p \rightarrow \bar{G}) \oplus (1 \rightarrow \bar{G}) = p$.

IV. 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 (where OP is · or +). Note that the exact form will vary with the type of structure as defined in the taxonomy of Section II.

Add Condition: A condition c_a that was not specified has been added to the implementation:

```
   if ($c_{i1}$ OP ... OP $c_{ij}$ OP...OP $c_{ini}$) then decision;
is replaced with
   if ($c_{i1}$ OP ... $c_{ij}$ OP $c_a$ OP $c_{ij+1}$ ...OP $c_{ini}$) then decision;
```

Delete Condition: A specified condition is missing from the implementation:

```
   if ($c_{i1}$ OP ... OP $c_{ij}$ OP...OP $c_{ini}$) then decision;
is replaced with
   if ($c_{i1}$ OP ... $c_{ij-1}$ OP $c_{ij+1}$ OP...OP $c_{ini}$) then decision;
```

Replace Condition: A condition is replaced with a different one, not equivalent:

```
   if ($c_{i1}$ OP ... OP $c_{ij}$ OP...OP $c_{ini}$) then decision;
is replaced with
   if ($c_{i1}$ OP ... $c_{ij-1}$ OP $c_x$ OP $c_{ij+1}$ OP...OP $c_{ini}$) then decision;
```

True Condition: A condition is always true:

```
   if ($c_{i1}$ OP ... OP $c_{ij}$ OP...OP $c_{ini}$) then decision;
is replaced with
   if ($c_{i1}$ OP ... OP true OP...OP $c_{ini}$) then decision;
```

False Condition: A condition is always false:

```
   if ($c_{i1}$ OP ... OP $c_{ij}$ OP...OP $c_{ini}$) then decision;
is replaced with
   if ($c_{i1}$ OP ... OP false OP...OP $c_{ini}$) then decision;
```

Negate Condition: A condition is negated:

```
   if ($c_{i1}$ OP ... OP $c_{ij}$ OP...OP $c_{ini}$) then decision;
is replaced with
   if ($c_{i1}$ OP ... OP ~$c_{ij}$ OP...OP $c_{ini}$) then ~decision;
```

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:

```
   if (condition) then decision;
is replaced with
   if (condition) then ~decision;
```

Delete Rule: A specified rule has been omitted from the implementation.

```
A rule from the policy is deleted:
   if ($c_{11}$ · . . . · $c_{1n1}$) then grant;
         . . .
   if ($c_{i1}$ · . . . · $c_{ini}$) then grant;
         . . .
   if ($c_{k1}$ · . . . · $c_{kn3}$) then grant;
   else deny;
is replaced with:
   if ($c_{11}$ · . . . · $c_{1n1}$) then grant;
         . . .
   if ($c_{k1}$ · . . . · $c_{kn3}$) then grant;
   else deny;
```

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. Tests can be produced by inserting flaws or mutations in a policy and then model checking or other methods to analyze the difference between the mutated and correct policies [3, 8, 14, 20].

V. Vulnerability Hierarchies

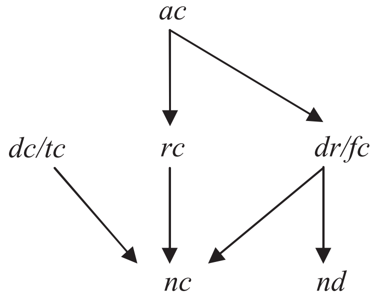
For the eight vulnerability classes introduced above, hierarchical relationships exist for the access control rule structures in the taxonomy of Sect. III. As can be seen, some common relationships emerge, but variations occur as a result of the differences in rule structure, e.g., conjunctions or disjunctions. In the structures below, vulnerabilities are abbreviated as follows:

ac: add condition
dc: delete condition
rc: replace condition
tc: true condition
fc: false condition
nc: negate condition
nd: negate decision
dr: delete rule

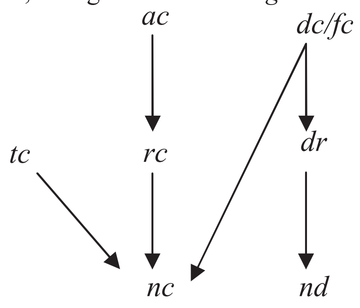
Note that this set of vulnerabilities is not claimed to be complete. If could be extended, for example, with vulnerabilities such as "add rule" or "replace rule". A more comprehensive collection will be studied in a future paper. Hierarchical relationships among detection conditions for n different vulnerability classes can be determined by checking for implications between the $n(n\text{-}1)$ 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. For instance, it was shown that $P \oplus P_{tc} \Rightarrow P \oplus P_{nc}$ for Example 2 above. For access control rules with a regular structure, such as those introduced in Section II, generic hierarchies can be constructed for various vulnerabilities.

The hierarchies below can be shown 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. (Computations not shown due to space limitation.) The analysis below is for policies containing a series of rules with grant decisions, followed by a default deny decision, where each rule has either a conjunction of two or more conditions (*con*/2/*singular*) or disjunction of two or more conditions (*dis*/2/*singular*). Similar hierarchies can be constructed for other rule patterns in the taxonomy of Sect. II. Although any propositional expression can be converted to a different but equivalent formula, the form of expressions must be considered when analyzing detection conditions because flaws in practice may not affect all occurrences of a variable. For example, $a + bc = (a+b)(a+c)$. If a is accidentally negated in one place on each side of the equation, the results are not equivalent: $\bar{a} + bc \neq (\bar{a}+b)(a+c)$.

con/2/*singular*, changes to rules with *grant* decision



dis/2/*singular*, changes to rules with *grant* decision



Note that for *con*/2/*singular* 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*, $abc \rightarrow G = 0bc \rightarrow G = 1 + G = 1$. Similarly, for *dis*/2/*singular*, dc and fc are equivalent, e.g., setting a to *false*, $a+b+c = 0+b+c = b+c$.

Example Policy

To illustrate the utility of the hierarchies discussed above, we define a small policy that could be implemented in a variety of frameworks such as XACML. The policy is summarized below, followed by a pseudo-code implementation which is then converted to a logic formula for analysis. **Policy:** *Supervisors may print customer account information during normal business hours or outside of business hours if special access is granted. Auditors may print customer information during normal business hours, or if authorized by a bank supervisor. Customer account information may be updated only by tellers through a teller terminal, or by accounting clerks if authorized by the accounting department supervisor.*

Implementation: This policy may be coded as follows:

```
if role = spvsr & busn_hrs & req = print → grant;
else if role = spvsr & !busn_hrs & spec_access & req = print
→ grant;
else if role = auditor & busn_hrs & req = print → grant;
else if role = auditor & !busn_hrs & spvsr_auth & req =
print → grant;
else if role = teller & node = teller_term & req = updat) →
grant;
else if role = acct & acct_spvsr_auth & req = update →
grant;
else deny;
```

Abbreviations for the terms in the policy are:

s: role = spvsr
b: busn_hrs
a: spec_access
p: req = print
d: role = auditor
h: spvsr_auth
t: role = teller
n: node = teller_term
u: req = update
c: role = acct
r: acct_spvsr_auth

Writing the policy as formula P for manipulation:

$$P = (sbp \rightarrow G)(s\bar{b}ap \rightarrow G)(dbp \rightarrow G)(d\bar{b}hp \rightarrow G)(tnu \rightarrow G)(cru \rightarrow G)(\sim(sbp) \sim(s\bar{b}ap) \sim(dbp) \sim(d\bar{b}hp) \sim(tnu) \sim(cru) \rightarrow \bar{G});$$

According to the taxonomy defined previously, this policy may be categorized as *con*/2/*singular* . Thus, as shown above, the relationships *dr/fc* → *nc* and *dr/fc* → *nd* must hold. That is, a test for either *delete rule* or *false condition* will detect both *negate condition* errors and *negate decision* errors.

A *negated condition* faulty policy occurs where the condition is negated. That is, if an administrator mistakenly codes the rule to specify the condition "req $\neq$ update" instead of "req = update", the policy is as follows:

$nc = (sbp \rightarrow G)(s\bar{b}ap \rightarrow G)(dbp \rightarrow G)(d\bar{b}hp \rightarrow G)(\boldsymbol{tn\bar{u}} \rightarrow G)(cru \rightarrow G)({\sim}(sbp)\ {\sim}(s\bar{b}ap)\ {\sim}(dbp)\ {\sim}(d\bar{b}hp)\ \boldsymbol{{\sim}(\ tn\bar{u})}\ {\sim}(cru) \rightarrow \bar{G});$

A *negate decision* error can occur in practice if a rule that should have a Grant decision is accidentally entered as Deny.

$nd = (sbp \rightarrow G)(s\bar{b}ap \rightarrow G)(dbp \rightarrow G)(d\bar{b}hp \rightarrow G)(\boldsymbol{tnu \rightarrow \bar{G}})(cru \rightarrow G)({\sim}(sbp)\ {\sim}(s\bar{b}ap)\ {\sim}(dbp)\ {\sim}(d\bar{b}hp)\ {\sim}(tnu)\ {\sim}(cru) \rightarrow \bar{G});$

In practice, a *delete rule* error may occur easily when a rule is accidentally removed or left out, particularly when policies contain dozens or hundreds of rules. As modeled, this would be written as:

$dr = (sbp \rightarrow G)(s\bar{b}ap \rightarrow G)\ (dbp \rightarrow G)\ (d\bar{b}hp \rightarrow G)$ **/*deleted rule*/** $(cru \rightarrow G)\ ({\sim}(sbp)\ {\sim}(s\bar{b}ap)\ {\sim}(dbp)\ {\sim}(d\bar{b}hp)\ {\sim}(cru) \rightarrow \bar{G});$

Detection conditions for each of these errors are as follows:

Detection conditions, negated condition: $dnc =$

$n \,\&\, t \,\&\, {\sim}c \,\&\, {\sim}p\ +$
$n \,\&\, t \,\&\, {\sim}p \,\&\, {\sim}r\ +$
$n \,\&\, t \,\&\, {\sim}p \,\&\, {\sim}u\ +$
$n \,\&\, t \,\&\, {\sim}c \,\&\, {\sim}d \,\&\, {\sim}s\ +$
$n \,\&\, t \,\&\, {\sim}d \,\&\, {\sim}r \,\&\, {\sim}s\ +$
$n \,\&\, t \,\&\, {\sim}d \,\&\, {\sim}s \,\&\, {\sim}u\ +$
$n \,\&\, t \,\&\, {\sim}a \,\&\, {\sim}b \,\&\, {\sim}c \,\&\, {\sim}d\ +$
$n \,\&\, t \,\&\, {\sim}a \,\&\, {\sim}b \,\&\, {\sim}c \,\&\, {\sim}h\ +$
$n \,\&\, t \,\&\, {\sim}a \,\&\, {\sim}b \,\&\, {\sim}d \,\&\, {\sim}r\ +$
$n \,\&\, t \,\&\, {\sim}a \,\&\, {\sim}b \,\&\, {\sim}d \,\&\, {\sim}u\ +$
$n \,\&\, t \,\&\, {\sim}a \,\&\, {\sim}b \,\&\, {\sim}h \,\&\, {\sim}r\ +$
$n \,\&\, t \,\&\, {\sim}a \,\&\, {\sim}b \,\&\, {\sim}h \,\&\, {\sim}u\ +$
$n \,\&\, t \,\&\, {\sim}b \,\&\, {\sim}c \,\&\, {\sim}h \,\&\, {\sim}s\ +$
$n \,\&\, t \,\&\, {\sim}b \,\&\, {\sim}h \,\&\, {\sim}r \,\&\, {\sim}s\ +$
$n \,\&\, t \,\&\, {\sim}b \,\&\, {\sim}h \,\&\, {\sim}s \,\&\, {\sim}u$

Detection conditions, negated decision: $dnd =$

$G \,\&\, n \,\&\, t \,\&\, u\ +$
$G \,\&\, n \,\&\, t \,\&\, u \,\&\, {\sim}p\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}c \,\&\, {\sim}p\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}p \,\&\, {\sim}r\ +$
$G \,\&\, n \,\&\, t \,\&\, u \,\&\, {\sim}d \,\&\, {\sim}s\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}c \,\&\, {\sim}d \,\&\, {\sim}s\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}d \,\&\, {\sim}r \,\&\, {\sim}s\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}a \,\&\, {\sim}b \,\&\, {\sim}c \,\&\, {\sim}d\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}a \,\&\, {\sim}b \,\&\, {\sim}c \,\&\, {\sim}h\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}a \,\&\, {\sim}b \,\&\, {\sim}d \,\&\, {\sim}r\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}b \,\&\, {\sim}c \,\&\, {\sim}h \,\&\, {\sim}s\ +$
$G \,\&\, n \,\&\, r \,\&\, t \,\&\, u \,\&\, {\sim}a \,\&\, {\sim}b \,\&\, {\sim}h\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}G \,\&\, {\sim}a \,\&\, {\sim}b \,\&\, {\sim}h \,\&\, {\sim}r\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}G \,\&\, {\sim}b \,\&\, {\sim}h \,\&\, {\sim}r \,\&\, {\sim}s$

Detection conditions, deleted rule: $ddr =$

$n \,\&\, t \,\&\, u \,\&\, {\sim}c \,\&\, {\sim}p\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}p \,\&\, {\sim}r\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}c \,\&\, {\sim}d \,\&\, {\sim}s\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}d \,\&\, {\sim}r \,\&\, {\sim}s\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}a \,\&\, {\sim}b \,\&\, {\sim}c \,\&\, {\sim}d\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}a \,\&\, {\sim}b \,\&\, {\sim}c \,\&\, {\sim}h\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}a \,\&\, {\sim}b \,\&\, {\sim}d \,\&\, {\sim}r\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}a \,\&\, {\sim}b \,\&\, {\sim}h \,\&\, {\sim}r\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}b \,\&\, {\sim}c \,\&\, {\sim}h \,\&\, {\sim}s\ +$
$n \,\&\, t \,\&\, u \,\&\, {\sim}b \,\&\, {\sim}h \,\&\, {\sim}r \,\&\, {\sim}s$

In other words, conditions that trigger a *deleted rule* vulnerability for the rule in the rule $tnu \rightarrow grant$ will also trigger a *negated condition* or a *negated decision* vulnerability. Because this implication relationship holds, a test for vulnerability *dr* will also detect vulnerability *nc*. Note that several tests occur in all three sets of detection conditions. For example $n \,\&\, t \,\&\, u \,\&\, {\sim}c \,\&\, {\sim}d \,\&\, {\sim}s$ is a single test that will detect policies that have been incorrectly containing a negated condition fault for u in the rule $tnu \rightarrow grant$, a negated decision fault for this rule, or if the rule has been left out completely.

Similarly, tests for *add condition* faults will detect four fault classes: *rc, dr, fc*, or *nd*. Thus tests must be generated for only two of the eight fault classes, because the other five types of faults will be detected by the generated tests as well. Note that this optimization works for policy specifications that fall into the *con/2/singular* category. For other configuration structures, different relationships hold, thus it is important to understand the type of configuration defined for a particular system. The analysis method described in Sect. V may be applied to any rule configuration structure, and is not confined to those specified in this paper.

Considering vulnerability hierarchies in test design can increase test efficiency by reducing the number of tests required to detect common errors, without reducing fault detection effectiveness, as can occur with other test minimization procedures [19]. Access control policy configuration is a problem that is particularly well suited to this type of analysis. Policies typically have a well-defined structure. Consequently, the set of common errors may be more restricted than, for example, may occur in ordinary programming. Conditions in rules may be omitted accidentally, placed in the wrong rule (added accidentally to one rule and left out of the other), the wrong decision may be specified for a particular set of conditions, coded as `variable = value` when `variable != value` was intended, and so forth. Using the hierarchies shown above, tests may be designed to detect common errors using fewer tests than would be required without knowledge of the hierarchical relationship among common errors. For example, if the policy includes *n* rules with *k* conditions each in the *con*/2/*singular* pattern, then no more than *n* tests for the deleted rule fault class will be needed to detect a deleted rule, negated decision, or if any of the *nk* conditions have been accidentally negated.

VI. Related Work

A key aspect of methods described here is the use of fault model for common errors in access control policies. A number of authors have reviewed access control policy faults for use in mutation testing, including [5,7,14,18,20]. Some of the vulnerability classes detailed in Sect. IV can be mapped to faults from fault models in these papers. This paper applies the methods for fault hierarchy construction developed for logic faults in boolean expressions to the problem of analyzing access control vulnerabilities introduced by configuration errors. Fault hierarchies in logic specifications were introduced in [11], and subsequently extended by others, including [9, 10, 12,17, 21]. These methods have not been previously applied to analysis of access control configurations. Several of the fault classes for boolean expressions apply also to vulnerability hierarchies, such as *stuck at true* or *stuck at false* faults, or *literal insertion faults*, which correspond to the *added condition* faults of Section IV, but others are unique to access control rules as a result of their structure. The *deleted rule* and *negated decision* faults discussed in this paper are two examples, and others could be developed as well. Early work on fault hierarchies assumed specifications were in disjunctive normal form, but hierarchies can still be shown after removing this assumption [17]. The DNF assumption is significant because DNF expressions are not commonly used in programming conditionals, and a fault in a general boolean expression can result in multiple faults when the expression is converted to DNF. Unlike conventional programming practice, access control system rules tend to be highly structured. This paper explicitly takes the form of various rule logic structures into account in computing vulnerability class hierarchies.

VII. Conclusions

This paper has demonstrated the application of methods for analyzing fault hierarchies to understanding relationships among vulnerabilities in access control rule structures. A taxonomy of rule structures was introduced and detection conditions computed for each class of vulnerability in the different structures. For two configuration structures, detection conditions were analyzed for the existence of logical implication relations between conditions. It was shown that hierarchies of detection conditions exist, and that hierarchies vary among rule structures. The existence of such hierarchies can be used to reduce the number of tests required to detect the presence of fault classes in access control rules as implemented. Other structures in the taxonomy will be analyzed in a forthcoming paper.

Acknowledgments: I am grateful to Lee Badger, Vincent Hu, and the Safeconfig reviewers for many helpful recommendations.

Certain software products are identified in this document, but such identification does not imply recommendation by NIST, or that the products identified are necessarily the best available for the purpose.

References

1 A. Anderson, Evaluating XACML as a Policy Language, OASIS Tech. Rpt. March, 2003.

2 R. Anderson, *Security Engineering, a Guide to Building Dependable Distributed Systems*, Wiley, 2001.

3 Paul E. Black, Vadim Okun, Yaacov Yesha, "Mutation Operators for Specifications," 15th *IEEE International Conf. Automated Software Engineering* (ASE'00), 2000, pp.81.

4 Center for Strategic and Intl Studies, “Securing Cyberspace for the 44th Presidency”, Dec. 2008.

5 K. Fisler, S. Krishnamurthi, L.A. Meyerovich, M.C. Tschantz, Verification and Change Impact Analysis of Access Control Policies, ACM, *Proc. 27th Intl Conf Software engineering 2005*, St. Louis, MO. pp. 196 – 205. ISBN:1-59593-963-2

6 V. Hu, D.R. Kuhn, T. Xie, J. Hwang, “*Model Checking for Verification of Mandatory Access Control Models and Properties*”, Int'l Journal of Software Engineering and Knowledge Engineering (IJSEKE), vol. 21, no. 1, pp. 103-127.

7 Hongxin Hu and GailJoon Ahn. Enabling verification and conformance testing for access control model. *Proc. 13th ACM Symposium on Access control Models and Technologies*, pages 195–204, Estes Park, CO,USA, June 2008.

8 J.H. Hwang, T. Xie, V.C. Hu, “Detection of Multiple-Duty-Related Security Leakage in Access Control Policies”, *3rd IEEE International Conference on Secure Software Integration and Reliability Improvement,* Shanghai, China, pp. 59-68, July 2009.

9 G. Kaminski, G. Williams, P. Ammann, “Reconciling Perspectives of Software Logic Testing”, *Software Testing, Verification, and Reliability*, vol. 18, pp. 149-188, 2008.

10 G. Kaminski, P. Ammann, Using a Fault Hierarchy to Improve the Efficiency of DNF Logic Mutation Testing, *Intl Conf Software Testing Verification and Validation*, 2009, 386-395.

11 D.R. Kuhn, "Fault Classes and Error Detection Capability of Specification Based Testing," *ACM Transactions on Software Engineering and Methodology,* Vol. 8, No. 4 (October, 1999)

12 M.F. Lau, Y.T. Yu, An extended fault class hierarchy for specification-based testing, ACM *Transactions on Software Engineering and Methodology* (TOSEM) Volume 14 , Issue 3 (July 2005), pp. 247 – 276.

13 A.X. Liu, M.G. Gouda, Firewall Policy Queries, IEEE Tran. Parallel and Distributed Sys, v. 20 n. 6, June 2009, pp. 766-777.

14 E. Martin, T. Xie, and T. Yu. Defining and Measuring Policy Coverage in Testing Access Control Policies. In *Proc., 8th International Conference on Information and Communications Security* (*ICICS 2006*), Raleigh, NC, pp. 139-158, 2006.

15 Internet Engineering Task Force, Internet Security Glossary, RFC 2828, May, 2000. http://www.ietf.org/rfc/rfc2828.txt

16 OASIS-Open. XACML v3.0 Hierarchical Resource Profile, Version 1.0, Working Draft 7, 1 April 2009.

17 V. Okun, P.E. Black, Y. Yesha, “Comparison of Fault Classes in Specification Based Testing”, Information and Software Technology, Elsevier, 46(8):525-533, 15 June 2004.

18 A. Pretschner, T. Mouelhi, and Y. Le Traon, “Model-Based Tests for Access Control Policies,” in *Proc., 1st Intl Conf Software Testing, Verification, and Validation (ICST ’08).* Lillehammer, Norway: IEEE, 9-11 April 2008, pp. 338–347.

19 G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proc. Intl Conf Software Maintenance*, pp. 34–43, 1998.

20 Y. L. Traon, T. Mouelhi, and B. Baudry. Testing security policies: Going beyond functional testing. *ISSRE '07. The 18th IEEE In. Symp. Software Reliability*.

21 T. Tsuchiya, T. Kikuno, On fault classes and error detection capability of specification-based testing, *ACM Trans Software Engineering and Methodology* (TOSEM), v.11, n. 1, pp. 58 – 62.