Pseudo-exhaustive Testing of Attribute Based Access Control Rules

D. Richard Kuhn[1], Vincent Hu[1], David F. Ferraiolo[1], Raghu N. Kacker[1], Yu Lei[2]

[1] *National Institute of Standards and Technology*
Gaithersburg, MD 20899, USA
{kuhn,vhu,david.ferraiolo,raghu.kacker}@nist.gov

[2] *Computer Science & Engineering*
University of Texas at Arlington
Arlington, TX, USA
ylei@uta.edu

Abstract – **Access control typically requires translating policies or rules given in natural language into a form such as a programming language or decision table, which can be processed by an access control system. Once rules have been described in machine-processable form, testing is necessary to ensure that the rules are implemented correctly. This paper describes an approach based on combinatorial test methods for efficiently testing access control rules, using the structure of attribute based access control (ABAC) to detect a large class of faults without a conventional test oracle.**

Keywords- access control; attribute based access control; combinatorial testing; t-way testing; test automation

NOMENCLATURE

R_i	= antecedent of *i*th grant rule
T_j	= conjunction of attributes in a rule antecedent
k	= maximum number of attributes in any term
m	= number of grant rules
n	= number of attributes
p	= average number of attributes in terms
v	= number of attribute values for an attribute
C	= correct term within a rule antecedent
F	= faulty term within a rule antecedent
N	= number of rows in covering array
P	= policy as specified
P'	= policy as implemented

I. INTRODUCTION - ABAC

Attribute based access control (ABAC) [1] is a method of controlling authorization using rules that include a subject's *attributes*, along with attributes of system resources, and conditions in the environment. For example, a rule may allow access to a database if the subject's attributes include *employee* and *US_citizen*, where the rule for accessing a resource requires these attributes. This approach can be more flexible than traditional models such as role-based access control, because it is not necessary to develop a structure of users and resources in advance. The tradeoff for such flexibility is that it may be difficult or impossible to know, at a point in time, which users have access to which resources. ABAC policies can also be highly complex, with hundreds of attributes and a large number of rules. Although ABAC policies can be quite large, their rules have a regular structure – checking for the presence of specified sets of attribute values – that makes it possible to apply combinatorial methods to reduce the effort required for high-assurance testing. The need for testing can be especially acute in cases where the protected application is hosted by a third party, such as a cloud service provider, and the data owner cannot directly inspect the software used in the access control system.

This paper describes a method of testing for ABAC systems that is *pseudo-exhaustive*, which we define as exhaustive testing of all combinations of attribute values on which an access control decision is dependent. This approach is analogous to pseudo-exhaustive methods for testing combinational circuits [2], 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. To test ABAC systems [1][3], 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, ABAC in particular, 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 an ABAC system, 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:

$$\begin{aligned} &R_1 \rightarrow \textit{grant} \\ &R_2 \rightarrow \textit{grant} \\ &\ldots \\ &R_m \rightarrow \textit{grant} \\ &\textit{else} \rightarrow \textit{deny} \end{aligned}$$

which is equivalent to:

$$\begin{aligned} &R_1 \rightarrow \textit{grant} \\ &R_2 \rightarrow \textit{grant} \\ &\ldots \\ &R_m \rightarrow \textit{grant} \\ &(\sim R_1)\,(\sim R_2)\ldots(\sim R_m) \rightarrow \textit{deny} \end{aligned}$$

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:

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

In a typical ABAC installation, the boolean literals could be 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*.

II. Testing ABAC Implementations

Testing an ABAC system 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 $x_1,\ldots,x_n$, $P'(x_1,\ldots,x_n) = P(x_1,\ldots,x_n)$.

1	0	0	0	1	0	0	0	0	0	1	0	0	0	1	0
2	0	0	1	0	1	1	1	1	1	0	0	0	0	0	1
3	0	1	0	0	0	1	0	1	1	1	1	1	1	0	0
4	0	1	1	1	1	0	0	1	0	0	0	1	0	0	1
5	1	0	0	0	1	0	0	1	0	0	1	1	1	1	0
6	1	0	1	1	0	1	1	1	0	1	0	1	1	1	0
7	1	1	0	1	1	1	0	0	0	1	0	0	1	0	1
8	1	1	1	0	0	0	1	1	1	1	1	0	1	1	1
9	1	0	0	0	0	1	1	0	1	0	0	1	1	1	1
10	0	1	1	1	0	1	1	0	1	0	1	1	0	1	0
11	0	1	0	0	1	0	1	0	0	0	1	0	1	1	1
12	1	0	1	1	1	0	0	0	1	0	1	0	1	0	0
13	1	0	0	1	1	0	1	1	1	1	1	1	0	1	1
14	1	0	1	0	0	1	0	0	0	0	1	1	0	0	1
15	0	1	1	0	0	1	1	0	0	1	1	0	0	0	0
16	0	1	1	0	1	0	0	0	1	1	0	1	1	1	1
17	1	1	0	1	0	0	1	1	1	0	0	0	0	0	0
18	0	0	0	0	1	1	1	1	1	1	1	1	1	0	0
19	0	1	0	0	1	0	0	1	0	1	0	0	0	0	0
20	1	1	1	1	0	0	1	0	0	1	1	1	0	1	1
21	1	0	0	0	1	1	0	0	1	0	0	0	0	1	0
22	0	1	1	1	0	1	1	1	0	1	0	0	1	1	1

Figure 1. 3-way covering array of 15 boolean parameters

How should an ABAC system be tested? 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 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 take advantage of combinatorial methods [4][5]. To see the advantages of a combinatorial approach, refer to Figure 1, which shows a covering array of 15 boolean variables. A covering array is an *N* x *k* array of *N* rows and *k* variables. In every *N* x *t* subarray, each *t*-tuple occurs at least once. In software testing, each row of the covering array represents a test, with one column for each parameter that is varied in testing. Collectively, the rows of the array include every *t*-way combination of parameter values at least once. For example, Figure 1 shows a covering array that includes all 3-way combinations of binary values for 15 parameters. Each column gives the values for a particular parameter. It can be seen that any three columns in any order contain all eight possible combinations of the parameter values. Collectively, this set of tests will exercise all 3-way combinations of input values in only 22 tests, as compared with 32,768 for exhaustive coverage. The size of a *t*-way covering array of *n* variables with *v* values each is proportional to $v^t \log n$ [6][7]. For the example described in Sect. I, with five attributes and two possible decisions, there are $2^5 = 32$ possible rules. However, a covering array of all 3-way combinations contains only 12 rows. The number of variables for which all settings are guaranteed to be covered in a covering array is referred to as the *strength*; a 3-way array is of strength 3.

We will use covering arrays of attributes, in association with ABAC 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. The covering array does not contain all possible input configurations, but it will contain all *k*-way combinations of variable values. 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, an ABAC 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 [5][6], 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. Modern constraint solvers such as Choco [8] and Z3 [9] make it possible to process very complex constraint sets, converting logic expressions into combinations that are invalid and can be screened from the final array.

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 $ac\overline{d}$, ae, and $\overline{a}b\overline{c}$. 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 ${\sim}R$.

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 ($ac\overline{d} + \overline{a}b\overline{c} \rightarrow grant$) 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, $ac\overline{d}$ and $\overline{a}b\overline{c}$, have been excluded from the array, but all other 1-, 2-, and 3-way combinations can be found in the array. Because $ac\overline{d}$ and $\overline{a}b\overline{c}$ 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*.

	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 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	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

Table 2. 3-way covering array with constraint ~R

III. Fault Detection

Now consider the faults that this method can detect. Suppose that some combination of attributes exists that produces a different response than required by the policy *P* in *k*-DNF form. Tests contained in GTEST and DTEST will detect a large class of missing terms, added terms, or altered terms containing *k* or fewer attributes. In this section we analyze faults that will be detected, and the underlying conditions in these faults. Table 3 illustrates the fault types and detection conditions for each.

	Term	**C=correct term**	**F=faulty term**	**GTEST detect condition**	**DTEST detect condition**	notes
1	**missing**	abc	--	abc	*none*	
2	**added**	--	ab	*none*	ab	
3		abc	$\bar{a}b$	*none*	$\bar{a}bc$, $\bar{a}b\bar{c}$	
4		abc	ab	*none*	$ab\bar{c}$	
5		ab	abc	--	--	*no fault*
6	**altered**	abc	$ab\bar{c}$	abc	$ab\bar{c}$	
7		abc	ab	*none*	$ab\bar{c}$	
8		abc	$\bar{a}b$	abc	$\bar{a}bc$, $\bar{a}b\bar{c}$	

Table 3. Example faults and detection conditions.

k-DNF detection property: Collectively, tests from GTEST and DTEST will detect added, deleted, or altered faults with up to *k* attributes.

Proof outline: Three cases can be considered, for missing, added, or altered terms in the policy. The analysis for each case is keyed to the numbered examples in Table 3.

Missing term. (1) Fault detected by GTEST. If a term is missing in the faulty implementation *P*', then there is some combination of attributes accepted in *P* that is not included in *P'*. Since it is in *P*, GTEST will include the combination and the fault will be detected.

Added term. If the system incorrectly issues a *grant* response, then some combination *F* of attributes accepted in *P'* is not included in *P*. We consider three cases depending on the number of attributes, *j*, in the added term.

> $j < k$ and *F* is not a subset of some other term (2, 3). Detected by DTEST because the added term will be part of the non-*grant* combinations in DTEST.
>
> $j < k$ and *F* is a subset of some other term (4). Detected by DTEST. If the *j* attributes of the added term are a subset of some term *C* in *P*, then because DTEST contains all *k*-way combinations not excluded by *R*, it will include all *k*-way combinations of the attributes in *F* with settings different from *C*. For example, if $F = ab = 11$, and $C = abc = 110$, then DTEST will include other settings of *abc*, which will include $abc = 111$. But because this term includes *F*, it will produce an incorrect grant response, detecting the fault. Note that if *P* contains both *abc* and $ab\overline{c}$, then the result of grant for $ab = 11$ is in fact correct, since $ab\overline{c} + abc = ab$.
>
> $j < k$ and there is some term *C* in *P* that is a subset of *F* (5). In this case a fault does not exist because any input that produces a *grant* response from *P* would produce a *grant* response with *F* added, because *F* contains all attributes of *C*.

Altered term. Three cases can be distinguished, based on the number of attributes *j* in the incorrect term *F* as compared with *k*.

> $j = k$ (6). Detected by GTEST because it includes a test with the correct term, and no other combination of attributes in that test will match any other term in *P*.
>
> $j < k$ and *F* is a subset of some other term (7). Detected by DTEST if there is no other term in *P*' that excludes from DTEST $F \cup x$, where *x* is one or more attributes in *C* that are not in *F*. Example: if $C = abc$ and *F* is *ab*, then $ab\overline{c}$ is in DTEST, unless *P'* also contains $b\overline{c}$. Note that if DTEST includes $F \cup x$, because there is some other term *D* in *P* where $x \subseteq D$, then there is no fault because the disjunction of the altered term with the other term would not accept any attribute sets not accepted in *P*. Not detected by GTEST because any test that contains the attributes of the correct term *C* will contain all attributes of a subset of *C*.
>
> $j < k$ and *F* is not a subset of altered term *C* (8). Detected by GTEST because it will include *C*, and *P'* will not match *C*. □

If more than k attributes are included in the altered term, some faults are still detected.

> $j > k$ and *C* is not a subset of *F*. Detected by GTEST because *C* will be included in GTEST but will produce a deny response.
>
> $j > k$ and *C* is a subset of *F*. Not detected by DTEST because DTEST excludes *C*, and therefore excludes *F* because it contains *C*. Not necessarily detected by GTEST because the settings of attributes *x* in *F* but not *C* may result in $C \cup x = F$. This case can be resolved by strengthening the covering array DTEST, using an array of strength k+i to detect faulty terms with up to *i* additional attributes.

IV. Tradeoffs and Practical Considerations

The process scales easily to systems with a large number of attributes that must be included in access decisions. Because the number of rows in a covering array grows only with log *n* for *n* variables/attributes at a given number of attributes and values, a larger policy specification, involving many more attributes, requires only a few additional tests. For example, it is possible to cover all 3-way combinations of 100 boolean variables with 45 tests, increasing only to 57 tests for 300 variables.

The most significant limitation for this approach occurs where terms in access control rules contain a large number of attribute values per attribute. Although covering array size grows only with log *n*, the value of *k* for the *k*-DNF form of rules is an exponent in the number of combinations that must be covered, and consequently the number of rows increases with v^k, for *v* attribute values. If terms in the rules contain more than six or seven attributes, it may not be practical to generate covering arrays, given the limitations of today's algorithms. However, a large number of tests is not a barrier, because the structure of the solution resolves the oracle problem by ensuring that every test in GTEST should produce a response of *grant* and every test in DTEST should produce a response of *deny*. This means that tests can be fully automated, making it possible to execute a large number of tests.

Table 5 shows test set sizes for a variety of configurations, assuming a value of $p = 4$ terms per rule. The column *N* tests gives the size *N* of a covering array for k-way combinations of *n* attributes with *v* values each. The size of DTEST is $m \times N$, since there will be one covering array per rule tested. Note that the test time for even a large test set of more than 600,000 tests would be roughly 10 minutes, assuming a time of 1 ms per test. In addition, since the tests are independent, testing can be divided among as many processors as are available, so even millions of tests could be tractable.

k	*v*	*n*	*m*	N tests	#GTEST	#DTEST
3	2	50	20	36	80	720
			50		200	1800
		100	20	45	80	900
			50		200	2250
	4	50	20	306	80	6120
			50		200	15300
		100	20	378	80	7560
			50		200	18900
	6	50	20	1041	80	20820
			50		200	52050
		100	20	1298	80	25960
			50		200	64900
4	2	50	20	98	80	1960
			50		200	4900
		100	20	125	80	2500
			50		200	6250
	4	50	20	1821	80	36420
			50		200	91050
		100	20	2337	80	46740
			50		200	116850
	6	50	20	9393	80	187860
			50		200	469650
		100	20	12085	80	241700
			50		200	604250

Table 4. Test set sizes.

V. HIPAA PRIVACY EXAMPLE

The following text is an excerpt from Health Insurance Portability and Accountability Act (HIPAA) rules that specify when a health care organization must treat a patient's personal representative of as the individual [10]. (This policy fragment has been used in previous work on formal specification of natural language policies [11].) Typical cases include a parent making decisions on behalf of a child.

> *(g)(1) Standard: Personal representatives. As specified in this paragraph, a covered entity must, except as provided in paragraphs (g)(3) and (g)(5) of this section, treat a personal representative as the individual for purposes of this subchapter.*
>
> *(2) Implementation specification: adults and emancipated minors. If under applicable law a person has authority to act on behalf of an individual who is an adult or an emancipated minor in making decisions related to health care, a covered entity must treat such person as a personal representative under this subchapter, with respect to protected health information relevant to such personal representation.*
>
> *(3)(i) Implementation specification: unemancipated minors. If under applicable law a parent, guardian, or other person acting in loco parentis has authority to act on behalf of an individual who is an unemancipated minor in making decisions related to health care, a covered entity must treat such person as a personal representative under this subchapter, with respect to protected health information relevant to such personal representation, except that such person may not be a personal representative of an unemancipated minor, and the minor has the authority to act as an individual, with respect to protected health information pertaining to a health care service, if:*
>
> *(A) The minor consents to such health care service; no other consent 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 be treated as the personal representative; (B) The minor may lawfully obtain such health care service without the consent of a parent, guardian, or other person acting in loco parentis, and the minor, a court, or another person authorized by law consents to such health care service; or (C) A parent, guardian, or other person acting in loco parentis assents to an agreement of confidentiality between a covered health care provider and the minor with respect to such health care service.*

Step 1: Review the text to identify attributes or variables that must be considered in access rules. Attributes in statutes may represent existence of various documents signed by the parties, among other basic attributes such as age, citizenship, etc. For this example, we consider the rules specified in (g)(3)(i)(A), for cases in which a minor has the authority to act as an individual. Each attribute is given a short mnemonic name in the covering array. These are shown below by bracketing each attribute, with a variable name annotation:

(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 be treated as the personal representative : 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; **or** (C) A **{parent, guardian, or other person acting in loco parentis assents to an agreement of confidentiality : pc}** between a covered health care provider and the minor with respect to such health care service.

Step 2: Convert the text description to rules in *k*-DNF form, in this case, 3-DNF. This mapping is as shown in Table 5. Note that the "or" connector prior to clause (C) indicates a disjunction of the three clauses.

Text	**Attributes**
(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;	expression: mc && ~oc && ~mr
	attribute sets: {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;	expression: lo && (mc||cc||oc) = lo && mc || lo && c || lo && oc
	attribute sets: {lo, mc}, {lo, cc}, {lo, oc}
(C) A **{parent, guardian, or other person acting in loco parentis assents to an agreement of confidentiality : pc}**	expression: pc
	attribute sets: {pc}

Table 5. Mapping of text to attributes.

Step 3: Determine the maximum number of *AND* connectives in access rule conditions. In the HIPAA example, three attributes are conjoined in Sect. (g)(3)(i)(A): "*minor consents ...; no other consent to such health care service is required by law, ...; and the minor has not requested that such person be treated as the personal representative"*. Because the expression above is in 3-DNF, with a maximum of three attributes in conjunction, a 3-way covering array is sufficient to consider all relevant combinations of attribute values. Note that other practical cases may involve more than 3-way combinations of attributes in terms, but the process would be the same as in this illustrative example. Combining these terms, we have:

mc && ~oc && ~mr || lo && (mc || cc || oc) || pc → *grant*
=
mc && ~oc && ~mr || lo&&mc || lo&&cc || lo&&oc || pc → *grant*

Step 4:
GTEST: Generate tests for GTEST, with one test for each term and other terms false. Tests are shown in Figure 2.

GTEST

	mc	oc	mr	lo	cc	pc
1	1	0	0	0	0	0
2	1	0	1	1	0	0
3	0	1	0	1	0	0
4	0	0	0	1	1	0
5	0	0	0	0	0	1

DTEST

	mc	oc	mr	lo	cc	pc
1	0	0	0	0	0	0
2	0	0	1	1	0	0
3	0	1	0	0	1	0
4	0	1	1	0	0	0
5	1	0	1	0	1	0
6	1	1	0	0	0	0
7	1	1	1	0	1	0
8	0	0	0	1	0	0
9	0	0	0	0	1	0
10	1	0	1	0	0	0
11	1	1	0	0	1	0
12	0	1	1	0	1	0

Figure 2. Completed GTEST and DTEST arrays.

DTEST: Compute a covering array for DTEST for the value of *t* determined in Step 2, using *~R* as a constraint. For illustration purposes, we include a covering array of the six variables without constraints in Figure 3, followed by a covering array computed with the expression above as a constraint. Note that no terms from the expression above can be found in the constrained array. For example, because array (1) includes all 3-way combinations, mc && ~oc && ~mr is present, but in (2) it is not found, although all other 3-way combinations of these variables are present in the array (2).

(1) Covering array without constraint

	mc	oc	mr	lo	cc	pc
1	0	0	0	0	0	0
2	0	0	1	1	1	1
3	0	1	0	1	0	1
4	0	1	1	0	1	0
5	1	0	0	1	1	0
6	1	0	1	0	0	1
7	1	1	0	0	1	1
8	1	1	1	1	0	0
9	1	1	0	0	0	0
10	0	0	1	1	0	0
11	0	0	0	0	1	1
12	1	1	1	1	1	1

(2) Covering array with constraint

	mc	oc	mr	lo	cc	pc
1	0	0	0	0	0	0
2	0	0	1	1	0	0
3	0	1	0	0	1	0
4	0	1	1	0	0	0
5	1	0	1	0	1	0
6	1	1	0	0	0	0
7	1	1	1	0	1	0
8	0	0	0	1	0	0
9	0	0	0	0	1	0
10	1	0	1	0	0	0
11	1	1	0	0	1	0
12	0	1	1	0	1	0

Figure 3. DTEST array compared with unconstrained array.

VI. Related Work

A variety of methods have been introduced for testing ABAC systems, particularly those implemented using the Extensible Access Control Meta Language (XACML) [12]. Xu and Zhang provide a survey of these methods [16], which have generally been based on a fault model with mutation operators, or policy coverage. Popular tools for fault model based testing include Margrave [19], which uses a specialized model checker, and has been applied to XACML policies. It has also been used as the foundation for mutation-based test generators, including a redundant rule checking tool called Cirg, which supports mutation testing tool Targen, which has been shown to produce high levels of structural coverage [17][15][18]. Another, the Access Control Policy Testing tools ACPT [20], uses the NuSMV language and model checker to support testing a broad range of policies, including ABAC, role based access control, and customized policies specified by users in the ACPT tool. ACPT includes combinatorial coverage of inputs for tests, a feature also provided by the Simple Combinatorial Test Generation algorithm and its related tools [22][23].

Pseudo-exhaustive test methods for circuit testing have an extensive history of application [2]. While our method is not derived from these earlier approaches, it shares the basic notion of determining dependencies, partitioning according to these dependencies, and testing exhaustively the inputs on which an output is dependent. We have previously applied this notion to software testing in a more general form, using the observation that faults depend on a small number of inputs, by covering all 2-way to 6-way combinations of inputs [24].

VII. Conclusions

Correct implementation of access control requires that policies written in natural language are mapped to machine-enforceable rules, and that these rules are correctly implemented. If access control rules contain at most k boolean attributes per conjunction, for an expression in k-DNF, then a k-way covering array includes all possible settings of such terms. Thus for any possible combination of n inputs, only k, $k < n$, matter in determining the truth of the expression. In most applications, the number of conditions will be small. The number of rows in a k-way covering array of boolean variables is proportional to $2^k \log n$. Thus for any given value of k, even a large number n of attributes requires only a test set proportional to $\log n$ to determine access for all possible inputs. The structure of the access control problem makes it possible to construct two test sets, GTEST and DTEST, for which the expected result is always *grant* or *deny* respectively. This structure eliminates the need for a conventional test oracle, so several hundred thousand tests can be generated and run automatically in a few minutes.

The HIPAA worked example included in this paper shows that this process is practical for real-world use. We plan to continue developing the approach, extending it for special cases such as priorities among rule conditions. As ABAC becomes more widely used, the method may assist developers in ensuring that policies are implemented correctly, and meet organizational requirements.

References

[1] V. C. Hu et al., Guide to Attribute Based Access Control (ABAC) Definition and Considerations, NIST Special Publication 800-162, January 2014.

[2] McCluskey, E. J. (1984). Verification Testing: A Pseudoexhaustive Test Technique. *Computers, IEEE Transactions on*, *100*(6), 541-546.

[3] Hu, V. C., Kuhn, D. R., Xie, T., & Hwang, J. (2011). Model checking for verification of mandatory access control models and properties. *International Journal of Software Engineering and Knowledge Engineering*, *21*(01), 103-127.

[4] Kuhn, D. R., Kacker, R. N., & Lei, Y. (2010). SP 800-142. Practical Combinatorial Testing.

[5] ACTS Home Page, http:// csrc.nist.gov/acts/

[6] Y. Lei, R. Kacker, D.R. Kuhn, V. Okun, J. Lawrence, IPOG: A general strategy for t-way software testing. *14th international conference on the engineering of computer-based systems*, 2007, pp 549–556

[7] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton, "The AETG System: An Approach toTesting Based on Combinatorial Design," *IEEE Trans. Software Eng.*, 23(7):437-444,1997.

[8] Jussien, N., Rochart, G., & Lorca, X. (2008). Choco: an open source java constraint programming library. In *CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08)* (pp. 1-10).

[9] De Moura, L., & Bjørner, N. (2008). Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems* (pp. 337-340). Springer Berlin Heidelberg.

[10] United States Congress. Health Insurance Portability and Accountability Act; Pub.L. 104–191, 110 Stat. 1936, enacted August 21, 1996

[11] H. DeYoung, D. Garg, L. Jia, D. Kaynar, and A. Datta. Experiences in the logical specification of the HIPAA and GLBA privacy laws. In Proceedings of the 9th annual ACM Workshop on Privacy in the Electronic Society (WPES), 2010. Full version: Carnegie Mellon University Technical Report CMU-CyLab-10-007.

[12] The eXtensible Access Control Markup Language (XACML), Version 3.0, OASIS Standard, January 22, 2013, <URL: http://docs.oasis-open.org/xacml/3.0/xacml-3.0-core-spec-os-en.pdf

[13] Bertolino, A., Lonetti, F., & Marchetti, E. (2010, September). Systematic XACML request generation for testing purposes. In *Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on* (pp. 3-11). IEEE.

[14] Hu, V. C., Martin, E., Hwang, J., & Xie, T. (2007, July). Conformance checking of access control policies specified in XACML. In *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International* (Vol. 2, pp. 275-280). IEEE.

[15] Martin, E., & Xie, T. (2007, May). A fault model and mutation testing of access control policies. In *Proceedings of the 16th international conference on World Wide Web* (pp. 667-676). ACM.

[16] Xu, D., & Zhang, Y. (2014, June). Specification and Analysis of Attribute-Based Access Control Policies: An Overview. In *Software*

Security and Reliability-Companion (SERE-C), 2014 IEEE Eighth International Conference on (pp. 41-49). IEEE.

[17] Martin, E. (2006, October). Automated test generation for access control policies. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications* (pp. 752-753). ACM.

[18] Martin, E., & Xie, T. (2007, May). Automated test generation for access control policies via change-impact analysis. In *Proceedings of the Third International Workshop on Software Engineering for Secure Systems* (p. 5). IEEE Computer Society.

[19] Fisler, K., Krishnamurthi, S., Meyerovich, L. A., & Tschantz, M. C. (2005, May). Verification and change-impact analysis of access-control policies. In *Proceedings of the 27th international conference on Software engineering* (pp. 196-205). ACM.

[20] V. Hu, D.R. Kuhn, T. Xie, Property Verification for Generic Access Control Models, IEEE/IFIP International Symposium on Trust, Security, and Privacy for Pervasive Applications, Shanghai, China, Dec. 17-20, 2008.

[21] ACPT Home Page, http://csrc.nist.gov/groups/SNS/acpt/access_control_policy_testing.html

[22] Bertolino, A., Lonetti, F., & Marchetti, E. (2010, September). Systematic XACML request generation for testing purposes. In *Software Engineering and Advanced Applications (SEAA), 2010 36th EUROMICRO Conference on* (pp. 3-11). IEEE.

[23] Bertolino, A., Daoudagh, S., Lonetti, F., & Marchetti, E. (2012, April). Automatic XACML requests generation for policy testing. In *Software Testing, Verification and Validation (ICST), 2012 IEEE Fifth International Conference on* (pp. 842-849). IEEE.

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

[25] Working DRAFT Information technology - Next Generation Access Control –Generic Operations and Data Structures (NGAC-GOADS)), INCITS 499-2013, American National Standard for Information Technology, American National Standards Institute, April 2014.