Assessing Quality of Policy Properties in Verification of Access Control Policies

Evan Martin
Google, Inc.
Santa Monica, CA
eemartin@google.com

JeeHyun Hwang Tao Xie
Computer Science Department
North Carolina State University
Raleigh, NC
{jhwang4, txie}@ncsu.edu

Vincent Hu
National Institute of Standards and Technology
Gaithersburg, MD, USA
vincent.hu@nist.gov

Abstract

Access control policies are often specified in declarative languages. In this paper, we propose a novel approach, called mutation verification, to assess the quality of properties specified for a policy and, in doing so, the quality of the verification itself. In our approach, given a policy and a set of properties, we first mutate the policy to generate various mutant policies, each with a single seeded fault. We then verify whether the properties hold for each mutant policy. If the properties still hold for a given mutant policy, then the quality of these properties is determined to be insufficient in guarding against the seeded fault, indicating that more properties are needed to augment the existing set of properties to provide higher confidence of the policy correctness. We have implemented Mutaver, a mutation verification tool for XACML, and applied it to policies and properties from a real-world software system.

1. Introduction

Access control is one of the most fundamental and widely used security mechanisms for resources. It controls which principals such as users or processes have access to which resources in a system. A growing trend has emerged toward writing access-control-policy specifications in standardized, declarative languages such as XACML [1] and Ponder [6]. Implementing and maintaining these policies are important and yet challenging tasks, especially as access control policies become more complex and are used to manage a large amount of distributed and sensitive information. Identifying discrepancies between policy specifications and their intended function is crucial because correct implementation and enforcement of policies by applications are based on the premise that the policy specifications are correct. As a result, policy specifications must undergo rigorous verification and validation to ensure that the policy specifications truly encapsulate the desires of the policy authors.

Property verification [9, 11, 13, 14, 19, 24] consumes a policy and a property, and determines whether the policy satisfies the property. Policy verification, while useful, requires the policy authors to write a property set, which can be verified against the policy under verification to ensure its correctness. With a property set of higher quality (covering a larger portion of a policy's behavior), the policy authors are more likely to detect policy faults (if any) and increase the confidence of its correctness.

In this paper, we propose a novel approach that assesses the quality of properties for a policy based on mutation verification, a counterpart of mutation testing [7] in verification. We have implemented our approach in a tool called Mutaver. To the best of our knowledge, there is no previous work to assess the quality of a property set and guide how to write a property set. The results of our mutation verification approach can be immediately used to aid property elicitation and serve as a general quality metric for a set of properties that ultimately check for faults in the policy.

In our approach, we propose *mutation verification* as a means to determine which properties in the given property set interact with rules in a policy during policy verification. In particular, given a policy, our approach automatically seeds it with faults to produce various mutant policies, each containing one fault. Then given a property set for this policy, our approach conducts property verification on this policy (called the original policy) and each mutant policy. If the property set holds for the original policy but fails to hold for the mutant policy, then the mutant is said to be killed by the property set. The ratio of the number of killed mutants to the total number of mutants serves as a metric to quantify comprehensiveness of the elicited property set. By analyzing the verification results, we can determine what rules, if any, fail to interact with the given property set and thus help guide property elicitation by targeting not-covered rules.

Different from previous research [17, 21] on policy mutation testing, instead of assessing the quality of a *request* set (by using a policy evaluation engine) in policy testing, we assess the quality of a *property* set by using a policy verification approach. A property set can effectively summarize various complex behaviors of a policy. In practice, a property is often intuitive and expressed in various ways (implicitly or explicitly) for a policy. Furthermore, as the behavior characterized by a property cannot be easily characterized by one or multiple policy requests, we cannot assess the property set with the previous approaches [17, 21] on policy mutation testing.

This paper makes the following main contributions:

- We propose a novel approach for assessing the quality of properties for a policy in policy verification.
- We implement the proposed approach with an automatic tool that facilitates automated mutation verification of access control policies written in XACML [1].
- We present a case study on an access control policy from a real-world software system to demonstrate the feasibility of this approach.

The rest of the paper is organized as follows. Section 2 presents an example, Section 3 offers some background information, Section 4 presents our approach, Section 5 describes our experiences of applying our approach on a real-world policy, Section 6 discusses issues in the approach, Section 7 presents related work, and Section 8 concludes.

2. Example

This section illustrates our approach to mutation verification through a simple example. The example and corresponding properties come from an example used by Fisler et al. [9]. This access control policy formalizes a university's policy on assigning and accessing grades. It is a role-based access control [8] policy with two roles specified in the subject attribute: Faculty and Student, two possible resource attributes: InternalGrades and ExternalGrades, and three possible action attributes: Assign, View, and Receive. For this example, we expect the following properties to hold:

Pr_1 There do not exist members of Student who can Assign ExternalGrades.

Pr_2 All members of Faculty can Assign both InternalGrades and ExternalGrades.

Pr_3 There exists no combination of roles such that a user with those roles can both Receive and Assign the resource ExternalGrades.

Property Pr_1 is intuitive since we certainly do not want students to assign grades. Property Pr_2 is to ensure that indeed faculty members can assign grades (otherwise who would assign them?). Finally, Pr_3 is an example of separation-of-duty since we do not want anyone to assign their own grade, an apparent conflict of interest.

```
If role = Faculty
  and resource =
(ExternalGrades or InternalGrades)
  and action = (View or Assign)
Then
  Permit
If role = Student
  and resource = ExternalGrades
  and action = Receive
Then
  Permit
```

Figure 1. Rules in an example XACML policy.

```
If role = Faculty
  and resource =
(ExternalGrades or InternalGrades)
  and action = (View or Assign)
Then
  Deny
If role = Student
  and resource = ExternalGrades
  and action = Receive
Then
  Permit
```

Figure 2. The first mutant XACML policy.

```
If role = Faculty
  and resource =
 (ExternalGrades or InternalGrades)
  and action = (View or Assign)
Then
  Permit
If role = Student
  and resource = ExternalGrades
  and action = Receive
Then
  Deny
```

Figure 3. The second mutant XACML policy.

Figure 1 shows the example XACML policy. To keep the example readable and concise, we write the policy as simple If-Then statements. This representation over-simplifies the complexity of XACML policies but suffices for illustrative purposes.

The first step of mutation verification is to generate mutant policies. For this example, we show only the mutants produced by the *Change Rule Effect (CRE)* mutation operator [17]. The CRE mutation operator simply inverts each rule's Effect by changing Permit to Deny, or Deny to Permit (one at a time for each mutant policy). The number of mutant policies created by this operator is equal to the number of rules in the policy. This operator should never create equivalent mutants, which are mutant policies with the same behavior as the original policy, unless a rule is unreachable. The example policy has only two rules and thus only two mutant policies are generated. Figures 2 and 3 show these two mutant policies for the first and second rules, respectively.

The second step of mutation verification is to determine which properties hold for the original policy and each mutant policy. The mutant is said to be killed by a property if the property holds for the original policy but does not hold for the mutant policy. In other words, the property reveals the fault seeded in the mutant policy. Similar to mutation testing [17, 21], the greater the number of mutants killed, the more comprehensive the properties are in covering policy behaviors, and thus the more effective the properties are at interacting with the rules in the policy.

The original policy (Figure 1) satisfies all three properties; therefore, if any property does not hold for a mutant policy, then that mutant policy is killed by the property.

The first mutant policy in Figure 2 does not satisfy Pr_2 and thus the first mutant is killed. Recall Pr_2 seeks to ensure that all faculty members can assign grades. Since the fault in Figure 2 is precisely the rule that grants this access, the property is apparently violated. The output of the property verification is a list of specific access requests that violate some property. The output from the property verification on the first mutant policy yields two requests: a request for a FACULTY to ASSIGN INTERNALGRADES and another request for a FACULTY to ASSIGN EXTERNALGRADES. Access is denied for both requests, indicating a violation of property Pr_2.

The second mutant policy in Figure 3 is not killed by any of the three properties, reflecting that the properties are not comprehensive and do not completely "cover" the policy. This realization leads to the elicitation of our fourth property, which was not originally specified by Fisler et al. [9]:

Pr_4 All members of STUDENT can RECEIVE EXTERNALGRADES.

Property Pr_4 fails to hold for the second mutant policy in Figure 3, thus killing the mutant, revealing its fault, and increasing the mutant-killing ratio.

In general, mutation verification serves two main purposes: (1) to quantify how thoroughly a set of properties interacts with or covers the policy behavior and (2) to facilitate property elicitation such that a property set interacts with or covers all rules defined in the policy. In particular, the CRE mutation operator is useful in identifying specific rules that are not covered by the property set. The CRE mutation operator and other mutation operators together are useful in quantifying the overall quality of the property set. As a by-product of this process, a test suite is generated consisting of each counterexample produced for each failing property in the form of a concrete access request and an expected response. The quality of this test suite is directly dependent on the quality of the property set.

3. Background

This section presents background information including a description of XACML, policy mutation testing, and Margrave, a policy verification tool used in our approach.

3.1. XACML

The eXtensible Access Control Markup Language (XACML) is an XML-based syntax used to express policies, requests, and responses. This general-purpose language for access control policies is an OASIS (Organization for the Advancement of Structured Information Standards) standard [1] that describes both a language for policies and a language for requests or responses of access control decisions. The policy language is used to describe general access control requirements and is designed to be extended to include new functions, data types, combining logic, etc.

3.2. Policy Mutation Testing

Mutation testing [7] has historically been applied to general-purpose programming languages. The program under test is iteratively mutated to produce numerous mutants, each containing one fault. A test input is independently executed on the original program and each mutant program. If the output of a test input executed on a mutant differs from the output of the same test input executed on the original program, then the seeded fault is detected and the mutant is said to be killed.

Policy mutation testing [17, 21] has been used to measure the fault-detection capability of a request set. In our previous work [17], we proposed a fault model for access control policies and defined a set of mutation operators that implement that model with the goal of programmatically creating mutant policies in order to evaluate test generation techniques and coverage criteria in terms of fault-detection capability. Similarly, Traon et al. [21] adapt mutation analysis and define mutation operators to quantify the effectiveness of a test set. Unfortunately, there are various expenses and barriers associated with mutation testing [17]. Primarily the generation and execution of a large number of mutants on a large test set. Fortunately, policy mutation testing is not as expensive as program mutation testing simply because policy specification languages are far simpler than general-purpose programming languages. Similarly and for the same reason, formal verification of policy specification is less costly. This distinction is one of the primary reasons that policy mutation verification is feasible. We use a variant of the policy mutation testing framework from our previous work [17] to facilitate the implementation of our policy mutation verification approach presented in Section 4.

Mutation analysis has been applied to model-based testing as well. Generally a model checker accepts a state-based model and a property, and outputs a counterexample if that property is not satisfied. The counterexample is essentially a test input that can then be executed on the concrete implementation of the model. Specification mutation [5] is a way to measure the effectiveness of a test input by mutating the specification. The specification is a set of properties that describe how the model should behave. By mutating the specification (i.e., properties), one can determine the adequacy of the model and its corresponding implementation. Conversely one can mutate the model [4, 10] to determine the adequacy of the specification. Our approach presented in Section 4 is analogous to model mutation where the model is the policy and the specification are the properties.

4. Mutation Verification

This section presents our approach for policy mutation verification to assess the quality of policy properties. We next describe the details of each step in the approach: mutant generation, property verification, mutant-killing determination, and property generation.

4.1. Mutant Generation

Given a policy, the first step is to generate a set of mutant policies. In our previous work [17], we presented a fault model for access control policies and a mutation testing framework to investigate the fault model. The framework includes mutation operators used to implement the fault model, mutant generation, equivalent-mutant detection, and mutant-killing determination. Previously, we used mutation testing to measure the quality of a request set in terms of fault-detection capability. In our new approach, we generate mutants not to measure the quality of a request set, but to measure the quality of a set of properties used for property verification. We use the *Change Rule Effect (CRE)* [17] mutation operator to help guide property elicitation to improve upon the existing property set.

The inputs to this step are the policy under test and a set of mutation operators. The mutator then generates a set of mutant policies, each with a single fault. Among generated mutants, semantically equivalent mutants to the original policy may exist. Such mutants and the original policy produce the same policy decisions for all possible requests. We leverage a change-impact analysis tool such as Margrave [9] to detect equivalent mutants by comparing the original policy and each mutant policy. Note that if a mutant is lengthy and semantically quite different from the original one, their comparison is often costly.

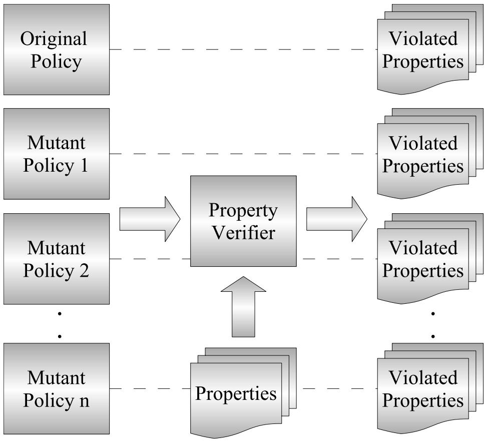

Figure 4. Property verification.

4.2. Property Verification

Given a policy, a set of properties, and a set of mutant policies, the next step is to determine which properties hold and which properties do not hold for both the original policy and each mutant policy as illustrated in Figure 4. We leverage Margrave [9] to perform property verification. Margrave represents `XACML` policies as multi-terminal binary decision diagrams (`MTBDDs`). `MTBDDs` are a type of decision diagram that maps bit vectors over a set of variables to a finite set of results. Margrave is implemented on top of the `CUDD` package [20], which provides an efficient implementation of `MTBDDs`. Margrave can verify various properties (that are represented in Margrave's specific format) against a given policy. Property Pr_i in Section 2 can be converted to Margrave's specific format and verified whether Pr_i is satisfied by the policy.

To perform property verification on a policy using Margrave, a Scheme program is written that leverages the Margrave API. This program must load the policy, optionally specify any environment constraints, and define the set of properties that the policy must satisfy. In order to perform property verification programmatically for each mutant policy, we generate an executable script and Scheme program for both the original policy and each mutant policy. We specified and generated an EMF[1] (Eclipse Modeling Framework) model that encapsulates the necessary information to generate the executable scripts and Scheme programs. Given an instance of this EMF model, we use the Model To Text (M2T[2]) framework and a set of Java Emitter Templates

[1] `http://www.eclipse.org/modeling/emf/`
[2] `http://www.eclipse.org/modeling/m2t/`

(JET) to generate executable shell scripts and Scheme programs for each policy under test and its corresponding set of mutants. The generated scripts pipe a generated Scheme program to a command-line Scheme interpreter. The output of the Scheme interpreter is then piped to a trace file for further processing. These trace files contain the information necessary for determining which properties hold and which properties do not hold for the original policy and each mutant policy.

4.3. Mutant-Killing Determination

The next step 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 quality of a set of properties with respect to covering a given policy. A higher mutant-killing ratio indicates the property set interacts with or covers a higher number of rules defined in the policy.

The trace files generated by the property verification described earlier are parsed in order to divide the property set into four subsets for each mutant. A property is either true or false (i.e., the property is satisfied or is not satisfied) with respect to a given policy. Let P denote the set of all properties. Let the set of properties satisfied by the original policy be denoted by O_T and let the set of properties not satisfied by the original policy be denoted O_F.

$$P = O_T \cup O_F \tag{1}$$

Similarly, given a mutant policy M, let the set of properties satisfied by the mutant policy be denoted by M_T and the set of properties not satisfied by the mutant policy be denoted M_F and by Equation 2 all properties fall into one of these sets.

$$P = M_T \cup M_F \tag{2}$$

For our case study described in Section 5, all properties are intended to be satisfied by the original policy (i.e., holding true). A property that holds true for both the original policy and the mutant policy cannot expose the fault in the mutant policy because the property does not apply to the portion of the policy that contains the fault. On the other hand, if at least one property holds true for the original policy but fails to hold true for the mutant policy as formalized in Equation 3, then the mutant is killed.

$$\exists p \in P : p \in O_T \cap M_F \tag{3}$$

4.4. Property Generation

To increase property coverage and investigate property verification, we manually generate additional properties for a given policy. A property Pr_i is elicited by considering a "living" CRE mutant. Each CRE mutant corresponds to a specific policy rule. When a given rule changes effect (e.g., deny to permit), the corresponding property Pr_i may detect a semantic change and by definition Pr_i kills the CRE mutant. To elicit Pr_i, we manually inspect the rule's context. Let S, O, and A denote the set of subjects, objects, and actions in a rule, respectively. A rule is in the form $[S \cap O \cap A \rightarrow Effect]$ where the effect is either $Permit$ or $Deny$. If an effect is $Permit$ ($Deny$), we generate a property that a subject S can (cannot) take an action A on resource O. For a mutant policy, when the effect is changed to $Deny$ ($Permit$), the property may detect a semantic fault in the mutated policy and cover the not-covered rule being mutated.

Table 1. Policies used in the case-study.

Subject	# Set	# Policy	# Rule	# Property
Continue-A	111	266	298	9
Continue-B	111	266	306	9
Simple-Policy	1	2	2	3

5. Case Study

We have applied our mutation verification tool to an access control policy for Continue [15]. Continue is a web-based conference management system that supports the submission, review, discussion, and notification phases of conferences. The Continue policy was used as a case study to explore property verification and change-impact analysis for Margrave by Fisler et al. [9]. The conference management system itself has been used to manage several conferences. Table 1 lists the policies used in our case study. Each row corresponds to a policy and Columns 2, 3, and 4 denote the number of PolicySet, Policy, and Rule elements in each policy, respectively. Column 5 denotes the number of properties used for each policy. We selected these policies in our case study because these policies are available with their formal properties. The Simple policy was presented in Section 2 and Continue-A and Continue-B are two versions of the Continue policy. All three policies and property sets are available at the Margrave web site[3].

We divide the case study into two parts: the first part performs mutation verification with the CRE mutation operator for property assessment while the second part performs mutation verification with several mutation operators and the additional manually specified properties for property augmentation.

[3] `http://www.cs.brown.edu/research/plt/software/margrave/`

Table 2. Mutant-killing ratios.

Subject	mutant-kill ratio	# mutants	# killed
CONTINUE-A	24.16%	298	72
CONTINUE-B	24.84%	306	76
SIMPLE-POLICY	50.00%	2	1

5.1. Property Assessment

Table 2 shows the results of mutation verification using only the CRE mutation operator, specifically the number of mutants (Column 3), number of killed mutants (Column 4), and the mutant-killing ratio (Column 2) for each policy. As discussed in Section 2, the SIMPLE policy has only two rules and thus two mutant policies are generated with the CRE mutation operator. One mutant is killed so the mutant-killing ratio is simply $\frac{1}{2}$ or 50%. The complexity of the CONTINUE policies makes them far more interesting. Each version of CONTINUE has approximately 300 rules and roughly $\frac{1}{4}$ of the mutants generated from these rules are killed. This result shows that a single property can cover multiple rules (which can be inferred by comparing the number of rules to the number of properties).

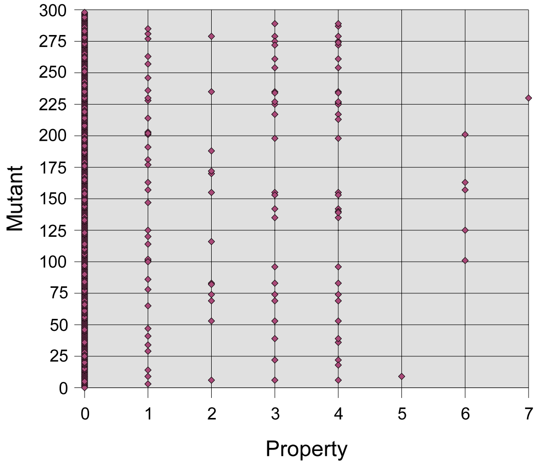

Figure 5. CONTINUE-A **property failures for each policy-property pair.**

To further visualize and discuss the results, let each property and each mutant be identified by an integer number. For example, let the original policy be denoted P_0, each mutant policy be denoted $P_1, P_2, \ldots P_m$, and each property $Pr_0, Pr_1, \ldots Pr_{p-1}$ where m and p are the number of mutants and properties, respectively. A property-policy pair (Pr_x, P_y) is mapped to a point (x, y) in Figures 5 and 6. A data point is plotted on the chart at (x, y) if the property Pr_x fails to hold for Policy P_y. Therefore, Figures 5 and 6 illustrate all property failures for each property-policy pair. More specifically, each integer value along the horizontal axis denotes a single property and each integer value along the vertical axis denotes a single policy. Furthermore, the policy at $y = 0$ is the original (un-mutated) policy. These scatter plots allow us to quickly determine which properties interact with which rules in the policy.

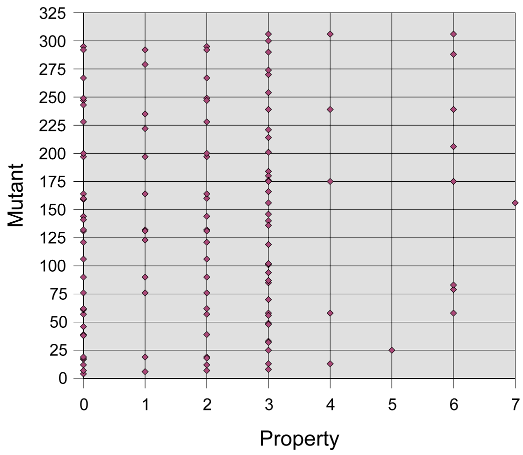

Figure 6. CONTINUE-B **property failures for each policy-property pair.**

Property Pr_0 fails to hold for the original version of CONTINUE-A indicated by a data point at (Pr_0, P_0) in Figure 5. As a result, Pr_0 also fails to hold for any mutant policies as indicated by the numerous data points along $x = 0$. If a property fails to hold for the original policy, then the property is not expected to (and most likely will not) hold for any mutant policies. Furthermore, this property is not capable of killing mutants because in order for a mutant to be killed by a property, the property must be satisfied by the original policy as described in Section 4. The natural language for this property is as follows:

Pr_0 If the subject is a pc-member, it is not the discussion phase, and unsubmitted for the review for a paper despite being assigned it, then the subject cannot see all parts of other's reviews for that paper.

This property fails simply because CONTINUE-A is an earlier version of the policy. All properties including Pr_0 do hold for the revised version in Figure 6.

Another readily noticeable peculiarity of both Figures 5 and 6 is the absence of Pr_8. Recall that nine properties are verified against each policy, implying one property, Pr_8, does not appear to interact with any rule explicitly defined in the policy. The natural language for the "missing" property is:

Pr_8 No legal request is mapped to Not Applicable, that is every legal request is decided by either deny or permit.

Pr_8 is an excellent example of a valid property that is not explicitly specified in the policy itself. A policy should certainly be written such that every legal request returns a deny or permit response. This property, however, is a generic property potentially applicable to a wide range of policies. Although the property is quite relevant, it is not (and arguably should not) be specified explicitly in the policy itself. An argument against its inclusion in the policy itself is that the property is generic; in particular, it is unrelated to the access control logic of the system but is rather one of the best practices. This type of generic property is not accounted for in this implementation of mutation verification. Further investigation is needed to determine how to incorporate such properties. For instance, mutation operators that consider not only the policy but also the properties may account for these types of properties.

The un-mutated CONTINUE-B (P_0 in Figure 6) satisfies all properties as indicated by the lack of data points along $P_0 = y = 0$. Again, Pr_8 (i.e., $x = 8$) is not plotted because this generic property does not interact directly with any rules specified in the policy. Properties Pr_5 and Pr_7 are interesting because they fail for only a single rule for both CONTINUE-A and CONTINUE-B. The natural language for these properties are:

Pr_5 If a subject is not a pc-chair or admin, then he/she may not set the meeting flag.

Pr_7 If someone is not a pc-chair or admin, then he/she can never see paper-review-rc for which he/she is conflicted.

By manual inspection, we determine that the mutant killed by Pr_5 is the same for both versions of the policy. The killed mutant corresponds to the last rule in the POLICYSET that specifies access to the meeting flag. More specifically, once all permitted combinations of subjects and actions are specified, the final rule ensures all other requests for the meeting flag are denied. Because the mutant policy changed this rule's decision to permit, the mutant was killed by Pr_5. In a similar fashion, the killed mutant for Pr_7 is identical for both versions and corresponds to precisely the rule that ensures the denial of requests for paper reviews when the isConflicted flag is set.

The CONTINUE policy heavily uses the first-applicable combining algorithm. As a result, it is often the case that, for a given resource, all permitting rules are specified first followed by more general denying rules. When these types of denying rules are mutated to permit, the policy leaks sensitive information (i.e., access is granted when it should not). A general property for ensuring that sensitive information remains protected is effective at identifying these leaks. For example, properties Pr_1 in Figure 5 and Pr_3 in Figure 6 are in fact the same property. This property interacts with a large number of policy rules indicated by the large number of data points. This property in natural language states that if the subject role attribute is empty and the resource class is not conference info, then return deny. This property effectively identifies information leakage introduced through the mechanism described earlier. This result indicates that this property set is effective at identifying information leakage in the policy.

On the other hand, the mutants that are not killed are generally those that mutate a permitting rule to deny. For example, when the rule that allows the admin to read the pcMember-info-isChairFlag is switched from permit to deny, no property identifies the restricted access. Similar to having general properties for ensuring that sensitive information remains protected, one also wants to have properties for ensuring access is granted when appropriate. The fact that the un-killed mutants are generally of this type indicates that the property set can be improved by adding properties for ensuring that access is granted when appropriate.

5.2. Property Augmentation

The low mutant-killing ratios in Section 5.1 imply that the existing property set can be improved by augmenting the existing property set. Motivated by these mutation verification results, we manually generate properties to kill the "living" CRE mutants. We do so by creating a property that mirrors a not-covered rule. Because each mutant created with the CRE mutation operator is associated with a single rule in the policy, we can identify which rules in the policy are not covered by the existing property set. After manually constructing these properties, we then perform mutation verification using various mutation operators to compare the quality of the existing property sets with the augmented property sets (i.e., the existing property set plus the manually specified properties).

Unfortunately, the size of the CONTINUE policy makes manually specifying properties impractical. As a result, for this case study, we investigate the first nine, commonly used rules in the CONTINUE-A and CONTINUE-B policies. Furthermore, because the difference between the CONTINUE-A and CONTINUE-B policies is small and the rules that we consider for manual property generation are identical, the results are the same for both policies. For brevity, we refer to the results of both policies simply as CONTINUE.

The manually generated properties are elicited by considering the "living" CRE mutants since each CRE mutant corresponds to a single rule in the policy. Since the SIMPLE policy has only one "living" mutant, only one additional property is specified. Furthermore, the property mirrors the policy rule. For example, the property "A student can write grades" is generated based on the policy rule "A student is permitted to write grades".

Table 3. Policy mutation and mutants killed by property sets.

	SIMPLE					CONTINUE				
Mutants	# mutants	# kill	kill %	# new kill	new kill %	# mutants	# kill	kill %	# new kill	new kill %
PSTT	0	0	0	0	0	2	2	100	2	100
PSTF	1	1	100	1	100	6	3	50	3	50
PTT	2	2	100	2	100	7	3	43	7	100
PTF	2	2	100	2	100	9	3	33	3	33
RTT	2	2	100	2	100	0	0	0	0	0
RTF	2	2	100	2	100	9	0	0	0	0
CPC	0	0	0	0	0	7	0	0	2	17
CRC	0	0	0	0	0	0	0	0	0	0
CRE	2	1	50	2	100	9	1	11	7	78
Total	11	10	90.91	11	100	49	12	24.49	24	48.98

Table 3 summarizes the number of generated mutants, the number of killed mutants, and the mutant-killing ratio for each policy for both the existing property set P and the augmented property set P_{new}. Each row of the table corresponds to a particular mutation operator and the column groups 1 and 2 correspond to the SIMPLE policy and the CONTINUE policy, respectively.

In the case study, we do not use some mutation operators if they result in equivalent mutants for the given policy or if the property verification tool cannot handle a particular XACML feature. For example, we omit RCT (Rule Condition True) and RCF (Rule Condition False) operations since the policies do not use that feature of rule conditions due to a limitation in the current Margrave. In addition, Margrave sometimes reports errors during property verification of some mutants. For instance, some properties require that certain elements exist in a given policy. Mutation operations such as PSTT (Policy Set Target True) may remove elements that are necessary in order to verify a particular property. These errors correctly indicate semantic faults in mutant policies and so we consider them killed.

The PTT (Policy Target True) and PTF (Policy Target False) operators delete or modify the top-level policy element in the SIMPLE policy effectively; these operators remove one of the two rules resulting in drastic semantic differences that are immediately detected by the both P and P_{new}. On the other hand, the CPC (Change Policy Combining Algorithm) and CRC (Change Rule Combining Algorithm) mutation operators generate equivalent mutants (i.e., mutants that are semantically equivalent to the original policy) that cannot be killed. For the CONTINUE policy, the CRC generates equivalent mutants. Such semantic equivalent mutants are detected and not considered in our case study.

Table 3 shows that the existing property set P can kill 50% and 11% of CRE mutants in SIMPLE and CONTINUE policies, respectively. By manually specifying a property that mirrors the not-covered rule for the SIMPLE policy, we can kill all CRE mutant policies. As expected, the augmented property set P_{new} increases mutant-killing ratios only in the CRE mutants since the remaining mutants cannot be killed. For the CONTINUE policy, we manually specify six properties and the augmented property set P_{new} kills 78% of the CRE mutants. The two not-covered rules (i.e., “living” CRE mutants) cannot be killed. P_{new} also increases the mutant-killing ratios for other mutation operators.

We observed that some types of mutants cannot be killed with P_{new}. For example, PSTT, CPC, and CRC mutants and RTT, RTF, and CRC mutants are not killed. These mutants may not be semantically equivalent to the original policy. However, the property set P_{new} is not sufficient to kill these mutants.

6. Discussion

We believe that our approach can be applied to assess the quality of a property set against policies written in languages other than XACML. Previous approaches converted policies in one language (such as XACML) to other languages (such as Alloy [12], *RW* [23], and Description Logics [14]) that are equipped with verification tools. As our approach requires property verification (against a policy and its mutants) provided by these verification tools, such conversion enables our approach to also be applicable to policies languages other than XACML and with verification tools other than Margrave.

Our approach to mutation verification provides a quality assessment of a property set for a policy. If a property set achieves a mutant-killing ratio of 100%, can we say that the property set is exhaustive or complete? This situation is similar to statement coverage in software testing. If a test suite achieves 100% statement coverage for a given program, can we say the test suite can detect all faults in the program? The answer, of course, is absolutely not. While mutation verification serves as a quality assessment for a property set and, with the CRE mutation operator, identifies which properties interact with which rules in the policy, it may not consider more abstract, generic properties. For example, Pr_1 of the illustrative example in Section 2 ensures that a student cannot assign grades. While this property is an intuitive one of the problem domain, it is not explicitly expressed in the policy itself. This particular policy contains only rules that *allow* access whereas this property is concerned with *denying* access. The fact that this property does not interact with the rules in the policy does not imply that it is not needed. A better example is discussed in Section 5 where a property serves as more of the best practice that is not related to the problem domain of the access control.

Future work shall investigate a means of automatically generating various types of properties to cover more rules and entities in an access control policy. In our case study, we manually generated properties to cover not-covered rules based on the mutation verification results for the CRE mutation operator. As we extracted these properties from (explicitly expressed) not-covered rules, each of the properties is specifically effective to kill the (previously un-killed) rule. But these properties may not kill other mutants. As existing properties often describe more general behaviors of a policy, further exploration of mutation operators for mutation verification is needed to investigate how to reflect relevant properties (that are not necessarily specified in the policy itself) in the mutation verification process.

7. Related Work

To the best of our knowledge, no metric has been defined to quantify the coverage of a policy or model by some property set. Our related previous approach on policy mutation testing [17] defined a fault model and corresponding automated mutator in order to quickly assess the quality of a test suite; the assessment results can be further used to assess test-generation and test-selection techniques in terms of fault-detection capability. Such policy mutation testing is related to the approach proposed by Ammann et al. [3] that mutates a model (corresponding to a policy in our work) and then uses the model mutants to assess the quality of a test suite. Our new approach leverages a variation of an automated mutator [17] in our implementation of the mutation verification approach. However, different from these previous approaches on assessing the quality of a test suite, our new approach focuses on assessing the quality of a property set based on mutating a policy.

To help ensure the correctness of policy specifications, researchers and practitioners have developed formal verification tools for policies. Several policy verification tools are developed specifically for firewall policies. Al-Shaer and Hamed [2] developed the Firewall Policy Advisor to classify and detect policy anomalies. Yuan et al. [22] developed the `FIREMAN` tool to detect misconfiguration of firewall policies.

There are also several verification tools available for `XACML` policies [1]. Hughes and Bultan [11] translated `XACML` policies to the Alloy language [12], and checked their properties using the Alloy Analyzer. Schaad and Moffett [19] also leverage Alloy to check that role-based access control policies do not allow roles to be assigned to users in ways that violate separation-of-duty constraints. Zhang et al. [24] developed a model-checking algorithm and tool support to evaluate access control policies written in *RW* languages, which can be converted to `XACML` [23]. Kolaczek [13] proposes to translate role-based access control policies into Prolog for verification. Kolovski et al. [14] formalize `XACML` policies with description logics (DL), which are a decidable fragment of the first-order logic, and exploit existing DL verifiers to conduct policy verification. Fisler et al. [9] developed Margrave, which can verify `XACML` policies against properties, if properties are specified, and perform change-impact analysis on two versions of policies when properties are not specified. When Margrave detects property violations during policy verification, it automatically generates concrete counterexamples in the form of specific requests that illustrate violations of the specified properties. Similarly, when Margrave detects semantic differences during change-impact analysis, it automatically generates specific requests that reveal semantic differences between two versions of a policy. Most of these approaches require user-specified properties to be verified. Our new approach complements these existing policy verification approaches because our approach helps assess the quality of the properties during policy verification.

Our previous work [16] proposed an approach to policy property inference via machine learning. Such properties are often not available in practice and their elicitation is a challenging and tedious task. Furthermore, once properties are defined, it is difficult to assess their effectiveness and identify potential problematic areas that need improvement. Our mutation verification approach intends to help alleviate that challenge. Our implementation leverages Margrave's property verification feature to verify mutant policies against properties.

Although various coverage criteria [25] for software programs exist, only recently have coverage criteria for access control policies been proposed [18]. Policy coverage criteria are needed to measure how well policies are tested and which parts of the policies are not covered by the existing test inputs. Martin et al. [18] defined policy coverage and developed a policy coverage measurement tool. Because it is tedious for developers to manually generate test inputs for policies, and manually generated test inputs are often not sufficient for achieving high policy coverage, they developed several test generation techniques. Different from these policy testing approaches, our new approach focuses on assessing the quality of properties in policy verification.

8. Conclusion

The need for carefully controlling access to sensitive information is increasing as the amount and availability of data are growing. In order to separate the semantics of access control from the system itself, policy authors increasingly specify access control policies in declarative languages such as XACML. Doing so facilitates managing, maintaining, and analyzing policies. To increase confidence in the correctness of specified policies, policy authors can formally verify policies against a property set. Policy verification is an important technique for high assurance of the correct specification of access control policies. Since the effectiveness of the verification process is directly dependent on the quality of the properties, we have proposed a novel approach called mutation verification to assess the quality of a property set in verification of access control policies. We have implemented a tool for the approach being applied on XACML policies. We applied our mutation verification tool to policies and properties from a real-world software system. Our experiences show that the performance of the property verification is encouraging and mutation verification can scale to sufficiently large access control policies. Furthermore, mutation verification is a complementary approach to property verification by aiding in the elicitation of properties.

Acknowledgment

This work is supported in part by NSF grant CNS-0716579 and its NIST supplement.

References

[1] OASIS eXtensible Access Control Markup Language (XACML). http://www.oasis-open.org/committees/xacml/, 2005.

[2] E. Al-Shaer and H. Hamed. Discovery of policy anomalies in distributed firewalls. In *Proc. INFOCOM*, pages 2605–2616, 2004.

[3] P. Ammann and P. E. Black. A specification-based coverage metric to evaluate test sets. In *Proc. HASE*, pages 239–248, 1999.

[4] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proc. ICFEM*, pages 46–54, 1998.

[5] T. A. Budd and A. S. Gopal. Program testing by specification mutation. *Computer Languages*, 10(1):63–73, 1985.

[6] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder policy specification language. In *Proc. POLICY*, pages 18–38, 2001.

[7] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[8] D. F. Ferraiolo, D. R. Kuhn, and R. Chandramouli. *Role-Based Access Control*. Artech House, Inc., 2003.

[9] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *Proc. ICSE*, pages 196–205, 2005.

[10] G. Fraser and F. Wotawa. Using model-checkers for mutation-based test-case generation, coverage analysis and specification analysis. In *Proc. ICSEA*, pages 16–21, 2006.

[11] G. Hughes and T. Bultan. Automated verification of access control policies. Technical Report 2004-22, Department of Computer Science, University of California, Santa Barbara, 2004.

[12] D. Jackson, I. Shlyakhter, and M. Sridharan. A micromodularity mechanism. In *Proc. ESEC/FSE*, pages 62–73, 2001.

[13] G. Kolaczek. Specification and verification of constraints in role based access control for enterprise security system. In *Proc. WETICE*, pages 190–195, 2003.

[14] V. Kolovski, J. Hendler, and B. Parsia. Analyzing web access control policies. In *Proc. WWW*, pages 677–686, 2007.

[15] S. Krishnamurthi. The CONTINUE server (or, how i administered PADL 2002 and 2003). In *Proc. PADL*, pages 2–16, 2003.

[16] E. Martin and T. Xie. Inferring access-control policy properties via machine learning. In *Proc. POLICY*, pages 235–238, 2006.

[17] E. Martin and T. Xie. A fault model and mutation testing of access control policies. In *Proc. WWW*, pages 667–676, 2007.

[18] E. Martin, T. Xie, and T. Yu. Defining and measuring policy coverage in testing access control policies. In *Proc. ICICS*, pages 139–158, 2006.

[19] A. Schaad and J. D. Moffett. A lightweight approach to specification and analysis of role-based access control extensions. In *Proc. SACMAT*, pages 13–22, 2002.

[20] F. Somenzi. CUDD: CU Decision Diagram Package. http://vlsi.colorado.edu/~fabio/CUDD/.

[21] Y. L. Traon, T. Mouelhi, and B. Baudry. Testing security policies: Going beyond functional testing. In *Proc. ISSRE*, pages 93–102, 2007.

[22] L. Yuan, J. Mai, Z. Su, H. Chen, C.-N. Chuah, and P. Mohapatra. FIREMAN: A toolkit for FIREwall Modeling and ANalysis. In *Proc. S&P*, pages 199–213, May 2006.

[23] N. Zhang, M. Ryan, and D. P. Guelev. Synthesising verified access control systems in XACML. In *Proc. FMSE*, pages 56–65, 2004.

[24] N. Zhang, M. Ryan, and D. P. Guelev. Evaluating access control policies through model checking. In *Proc. InfoSec*, pages 446–460, 2005.

[25] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.