Formalizing Semantic Differences between Combining Algorithms in XACML 3.0 Policies

Dianxiang Xu, Yunpeng Zhang, Ning Shen
Department of Computer Science
Boise State University
Boise, ID 83725, USA
{dianxiangxu, yunpengzhang, ningshen}@boisestate.edu

***Abstract*—XACML is a standard language for specifying attribute-based access control policies of computer and software systems. It offers a variety of combining algorithms for flexible policy composition. While they are intended to be different, they also bear similarities. Some combining algorithms can be functionally equivalent with respect to the given policy or policies. To correctly use the combining algorithms, it is important to understand the subtle similarities and differences. This paper presents a formal treatment of the semantic differences between the commonly used combining algorithms in XACML 3.0. For each pair of the selected combining algorithms, we identify when they are functionally equivalent and when they are not equivalent. This rigorous understanding helps minimize incorrect uses of combining algorithms that may lead to unauthorized access and denial of service. It also provides a foundation for determining equivalent mutants of combining algorithms in mutation testing of XACML policies.**

Keywords—Attribute-based access control, combining algorithm, formal methods, mutation testing, XACML.

I. INTRODUCTION

With the increasing complexity of computer software, access control methods have evolved from role-based to attribute-based authorization. Attribute-based access control (ABAC) [9] enables fine-grained access control by combining various attributes of authorization elements into access control decisions. These attributes are predefined characteristics of subjects (e.g., job title and age), resources (e.g., data, programs, and networks), actions, and environments (e.g., current time and IP address) [6]. ABAC also facilitates collaborative policy administration within a large enterprise or across multiple organizations [4][10]. In a large enterprise, for example, elements of authorization policies may be managed by different departments, such as the Information Technology department, Human Resources, the Legal department, and the Finance department [14]. Individual rules or policies are composed into a whole in order to make consistent access decisions. As an OASIS standard for specifying ABAC policies in the XML format, XACML (eXtensible Access Control Markup Language) [14] supports flexible composition of access control policies by predefining 11 rule combining algorithms and 12 policy combining algorithms. A combining algorithm aims at rendering a single access decision by combining the decisions of individual access control rules or policies.

Due to the variety of combining algorithms and subtle similarities between different combining algorithms, it is not uncommon to use combining algorithms incorrectly when XACML3.0 policies are authored. The standard specification of XACML3.0 lacks a rigorous representation of the semantics of language elements, particularly combining algorithms. This paper presents a formal treatment of the semantic differences between various rule and policy combining algorithms in XACML 3.0. While different combining algorithms are designed to be different, this paper shows that, for any pair of rule (or policy) combining algorithms we have studied, they can be functionally equivalent with respect to certain rules (or policies). As will be illustrated in Section III, for example, the rule combining algorithms *Deny-overrides* and *Permit-overrides* may make no difference with respect to a policy with certain permit and deny rules. Without such an insight, one would believe that these combining algorithms are different.

The formalization of the semantic differences between combining algorithms is a significant contribution for two reasons. First, a rigorous understanding helps minimize incorrect uses of combining algorithms, which can lead to unauthorized access and denial of service. A user may inadvertently select an incorrect combining algorithm or intentionally apply an incorrect combining algorithm due to misunderstanding. When different combining algorithms are functionally equivalent for certain rules (or policies), one may use them interchangably. In an evolving process of policy development and maintenance, however, a previously working combining algorithm may become incorrect after new rules or policies are added in a way that implicitly breaks the constraints on functional equivalence. Second, the formalization provides a theoretical foundation for determining equivalent mutants of combining algorithms in mutation testing of XACML policies. Existing working on testing XACML policies has commonly used mutation to evaluate proposed testing methods [2][3][12]. The idea is to create mutants of an XACML policy and run test cases against each mutant. A mutant is a variant of the original policy with one injected fault (e.g., the rule combining algorithm is changed from *Deny-overrides* to *Permit-overrides*). A mutant is said to be killed if at least one test reports a failure when the mutant is tested. The fault detection capability of the test cases is indicated by the mutation score, i.e., the ratio between the number of killed mutants and the total number of non-equivalent mutants. Obviously, mutation testing needs to determine whether each

978-1-4673-7989-2/15 $31.00 © 2015 European Union
DOI 10.1109/QRS.2015.32

live mutant is equivalent to the original policy. This research applies to policy mutants with changed combining algorithms.

The remainder of this paper is organized as follows. Section II introduces XACML policy elements. Section III focuses on the semantic differences between rule combining algorithms. Section IV applied the results to several policy patterns where multiple rule combining algorithms can be functionally equivalent and thus interchangeable. Section V discusses the semantic differences between policy combining algorithms. Section VI presents the case studies. Section VII reviews related work. Section VIII concludes this paper.

II. XACML Language Elements

The main components of the XACML3.0 model are rule, policy, and policy set. As the most elementary unit of policy, a rule *r*, denoted as *<rt, rc, re>,* consists of a target *rt*, a condition *rc*, and an effect *re*. The target *rt* is a logical expression that specifies the set of requests to which the rule is intended to apply. The condition *rc* is a Boolean expression that refines the applicability of the rule established by the target. The rule effect *re* is either *Deny* or *Permit*. *r* is called a permit rule if *re=Permit*; *r* is called a deny rule if *re=Deny*; *rt* and *rc* are optional. A rule without target and condition, denoted by *<_, _, re>* is called a default rule.

A policy *P*, denoted as *<PT, CA, R>,* comprises a policy target *PT*, a rule-combining algorithm identifier *CA*, and a list of rules *R*. A policy set *PS*, denoted as *<PST, PCA, PL>*, consists of a policy set target, a policy-combining algorithm identifier, and a list of policies or policy sets. Figure 1 shows the relationships between the main elements of XACML3.0.

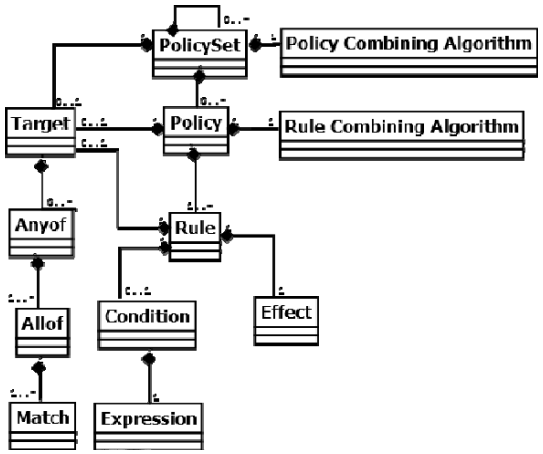

Figure 1. Main language elements of XACML 3.0

Policy target, rule target, and rule condition are predicates defined over attribute names and values. For example, *age≥18* is a predicate that describes a relation between attribute *age* and value 18. A query consists of a list of attribute assignments: *{x_1=V_1, x_2=V_2,...}*, where x_i is an attribute name and V_i is a value assigned to x_i. The decision of rule *r=<rt, rc, re>* with respect to query *q*, denoted by *d(r, q)*, is defined as follows:

- *Permit*: access is granted when rule effect *re* = *Permit*, query *q* matches policy target *PT* and rule target *rt*, and rule condition *rc* is true with respect to *q*.
- *Deny*: access is denied when *re* = *Deny*, *q* matches *PT* and *rt*, and *rc* is true with respect to *q*.
- *N/A*: *q* is not applicable – *q* does not match *rt* or *rc* evaluate to false with respect to *q*.
- *I(D)*: An error occurred when *rt* or *rc* was evaluated and *re=Deny*. The decision could have evaluated to *Deny* if no error had occurred.
- *I(P)*: An error occurred when *rt* or *rc* was evaluated and *re=Permit*. The decision could have evaluated to *Permit* if no error had occurred.

For convenience, here we use *N/A, I(D), I(P),* and *I(DP)* to denote the following decisions respectively: *NotApplicable*, *Indeterminate {D}*, *Indeterminate {P}*, and *Indeterminate {DP}*. So *d(r, q) ∈ {Permit, N/A, I(P)}* if *r* is a permit rule, and *d(r, q) ∈{Deny, N/A, I(D)}* if *r* is a deny rule. For a default rule *r = <_, _, re>*, *d(r, q) = re* for any *q*.

For a given query *q*, rules $r_1, r_2..., r_n$ in policy *P=<PT, CA, R>* may yield different decisions. The rule combining algorithm *CA* combines the decisions of individual rules into a single policy-level decision, denoted as *d(P, q)*. The set of possible policy decisions of *d(P, q)* depends on *CA*. For example, *Deny-overrides*, *Permit-overrides*, and *First-applicable* may yield one of the following six decisions: *{Permit, Deny, N/A, I(D), I(P), I(DP)}*[1], where *I(DP)* refers to *Indeterminate{DP}*. *I(DP)* results from one of the following situations: (a) an error occurred when policy target *PT* was evaluated and the decision could have evaluated to *Deny* or *Permit* if no error had occurred; (b) there is a permit rule that evaluates to *I(P)* and a deny rule that evaluates to *I(D)* or *Deny* when *CA=Permit-overrides;* (c) there is a deny rule that evaluates to *I(D)* and a permit rule that evaluates to *I(P)* or *Permit* when *CA=Deny-overrides*. *Deny-unless-permit* and *Permit-unless-deny* result in either *Permit* or *Deny*.

III. Comparing Rule Combining Algorithms

In XACML 3.0, there are 11 rule combining algorithms. Four are for compatibility support of old versions - *Legacy Ordered-deny-overrides*, *Legacy Permit-overrides*, *Legacy Ordered-permit-overrides*, and *Legacy Ordered-permit-overrides*. In Balana [1] (an open source implementation of XACML3.0 based on which our research is conducted), the implementations of *Ordered-deny-overrides* and *Ordered-permit-overrides* are the same as *Deny-overrides* and *Permit-overrides*. Thus, this paper focuses on five rule combining algorithms: *Deny-overrides, Deny-unless-permit, Permit-overrides, Permit-unless-deny,* and *First-applicable*. According to XACML 3.0 specification, their meanings are as follows:

[1] According to the XACML3.0 standard, *I(D)*, *I(P)* and *I(DP)* will be a plain *Indeterminate* if it is the final decision returned by PDP. When generating query *q* for *d(P, q) ≠ d(P', q)*, our approach tries to avoid the cases where the final decisions of *d(P, q)* and *d(P', q)* are both *Indeterminate*.

- *Deny-overrides*: Intended for those cases where a deny decision should have priority over a permit decision;
- *Permit-overrides*: Intended for the cases where a permit decision should have priority over a deny decision.
- *Deny-unless-permit*: Intended for those cases where a permit decision should have priority over a deny decision, and an "*Indeterminate*" or "*NotApplicable*" must never be the result.
- *Permit-unless-deny*: Intended for those cases where a deny decision should have priority over a permit decision, and an "*Indeterminate*" or "*NotApplicable*" must never be the result.
- *First-applicable*: Rules are evaluated in the order in which they are listed. If a rule's target matches and condition evaluates to "True", then return the rule's effect (*Permit* or *Deny*). If the target or condition evaluates to "False", the next rule is evaluated. If no further rule exists, then return "*NotApplicable*". If an error occurs, then return "*Indeterminate*", with the appropriate error status.

We have formalized the semantic differences between the five rule combining algorithms (i.e., 10 pairs) with 22 theorems. The details can be found in our technical report [15]. Due to the space constraint, this section first introduces *Deny-overrides* vs *Permit-overrides* and *Deny-overrides vs Deny-unless-permit* as examples. Then we summarize the main observations from the comparisons of all 10 pairs of the five rule combining algorithms.

To ensure complete comparison of rule combining algorithms, we usually classify all possible rules into disjoint categories and prove when two rule combining algorithms are functionally equivalent with respect to the given rules, and if not equivalent, what kind of queries reveals their difference.

A. Deny-overrides vs Permit-overrides

We classify all possible rules into three categories: *R* is composed of only permit rules, *R* is composed of only deny rules, and *R* has both deny and permit rules. For the first two categories, *Deny-overrides* and *Permit-overrides* are functionally equivalent, as shown in *Theorem 1* below. When *R* has both deny and permit rules, *Deny-overrides* and *Permit-overrides* may or may not be functionally equivalent, as shown in *Theorem 2* below.

Theorem 1. Given policy $P = <PT, Deny\text{-}overrides, R>$ and $P' = <PT, Permit\text{-}overrides, R>$. If r_i ($1 \le i \le n$) are all permit rules or r_i ($1 \le i \le n$) are all deny rules, then P and P' are functionally equivalent.

Proof: Let us first prove that $d(P, q) = d(P', q)$ for any query q if $r_i \in R$ ($1 \le i \le n$) are all permit rules.

(1) The theorem holds for one rule, i.e., $R = \{r_1\}$. For any q, $d(r_1, q) \in \{Permit, I(P), N/A\}$.
- If $d(r_1, q) = Permit$, then $d(P, q) = Permit$ and $d(P', q) = Permit$. So $d(P, q) = d(P', q)$.
- If $d(r_1, q) = I(P)$, then $d(P, q) = d(P', q) = I(P)$.
- If $d(r_1, q) = N/A$, then $d(P, q) = d(P', q) = N/A$.

(2) The theorem holds for two rules, i.e., $R = \{r_1, r_2\}$. Note that for any q, $d(r_1, q) \in \{Permit, I(P), N/A\}$, and $d(r_2, q) \in \{Permit, I(P), N/A\}$. The following table presents the results of both combining algorithms for any $d(r_1, q) \in \{Permit, I(P), N/A\}$ and $d(r_2, q) \in \{Permit, I(P), N/A\}$. So $d(P, q) = d(P', q)$ for any q.

TABLE I. DENY-OVERRIDES/PERMIT-OVERRIDES FOR TWO PERMIT RULES

	$d(r_1, q) = Permit$	$d(r_1, q) = I(P)$	$d(r_1, q) = N/A$
$d(r_2, q) = Permit$	Permit	Permit	Permit
$d(r_2, q) = I(P)$	Permit	I(P)	I(P)
$d(r_2, q) = N/A$	Permit	I(P)	N/A

(3) Suppose the theorem holds for *n-1* rules. $d(P_{n-1}, q) = d(P'_{n-1}, q)$, where $P_{n-1} = <PT, Deny\text{-}overrides, \{r_1, ..., r_{n-1}\}>$ and $P'_{n-1} = <PT, Permit\text{-}overrides, \{r_1, ..., r_{n-1}\}>$. We show that the theorem also holds for *n* rules. Note that for any q, $d(P_{n-1}, q) \in \{Permit, I(P), N/A\}$, and $d(r_n, q) \in \{Permit, I(P), N/A\}$. Similar to (2), the following table presents the results of both combining algorithms for any $d(P_{n-1}, q) \in \{Permit, I(P), N/A\}$ and $d(r_n, q) \in \{Permit, I(P), N/A\}$. In other words, $d(P, q) = d(P', q)$ for any q.

TABLE II. DENY-OVERRIDES/PERMIT-OVERRIDES FOR N(>2) PERMIT RULES

	$d(P_{n-1}, q) = Permit$	$d(P_{n-1}, q) = I(P)$	$d(P_{n-1}, q) = N/A$
$d(r_n, q) = Permit$	Permit	Permit	Permit
$d(r_n, q) = I(P)$	Permit	I(P)	I(P)
$d(r_n, q) = N/A$	Permit	I(P)	N/A

Similarly, we can prove $d(P, q) = d(P', q)$ for any q if r_i ($1 \le i \le n$) are all deny rules. To do so, we replace every occurrence of *Permit* with *Deny* and replace every occurrence of *I(P)* with *I(D)* in the above proof. So *Theorem 1* holds.

Theorem 2. Given policy $P = <PT, Deny\text{-}overrides, R>$ and $P' = <PT, Permit\text{-}overrides, R>$, where *R* has at least one permit rule and at least one deny rule. For any q, $d(P, q) \neq d(P', q)$ if and only if there exists permit rule $r_i = <rt_i, rc_i, Permit> \in R$, deny rule $r_j = <rt_j, rc_j, Deny> \in R$, and query q, such that:

(a) $d(r_i, q) = Permit \wedge d(r_j, q) \in \{Deny, I(D)\}$ or

(b) $d(r_i, q) = I(P) \wedge d(r_j, q) = Deny$.

Proof: (1) Sufficient condition

- If $d(r_i, q) = Permit \wedge d(r_j, q) = Deny$, then $d(P, q) = Deny$, and $d(P', q) = Permit$. Thus $d(P, q) \neq d(P', q)$.
- If $d(r_i, q) = Permit \wedge d(r_j, q) = I(D)$, then $d(P, q) \in \{Deny, I(DP)\}$, and $d(P, q) \neq Permit$. $d(P', q) = Permit$. Thus, $d(P, q) \neq d(P', q)$.
- If $d(r_i, q) = I(P) \wedge d(r_j, q) = Deny$. $d(P, q) = Deny$. $d(P', q) \neq Deny$. Thus $d(P, q) \neq d(P', q)$.

(2) Necessary condition. If there does not exist permit rule $r_i = <rt_i, rc_i, Permit> \in R$ and deny rule $r_j = <rt_j, rc_j, Deny>$ such that (a) and (b) hold. In other words, for each permit rule r_i and deny rule r_j: $\neg (d(r_i, q) = Permit \wedge d(r_j, q) \in \{Deny, I(D)\})$

$\wedge \neg (d(r_i, q) = I(P) \wedge d(r_j, q) = Deny)$. We convert this to a disjunctive normal form and then simplify it as follows:

$d(r_i, q) \neq Permit \vee d(r_j, q) = N/A)) \wedge$
$(d(r_i, q) \neq I(P) \vee d(r_j, q) \neq Deny))$
$= d(r_i, q) \neq Permit \wedge d(r_i, q) \neq I(P) \vee$
$d(r_j, q) = N/A \wedge d(r_i, q) \neq I(P) \vee$
$d(r_i, q) \neq Permit \wedge d(r_j, q) \neq Deny \vee$
$d(r_j, q) = N/A \wedge d(r_j, q) \neq Deny$
$= d(r_i, q) = N/A \vee$
$(d(r_i, q) \in \{I(P), N/A\} \wedge d(r_j, q) \in \{I(D), N/A\}) \vee$
$d(r_j, q) = N/A$
$= (d(r_i, q) \in \{I(P), N/A\} \wedge d(r_j, q) \in \{I(D), N/A\})$

When all permit (or deny) rules evaluate to *N/A*, the combined result depends on the other deny (or permit) rules. Similar to *Theorem 1*, *Deny-overrides* and *Permit-overrides* make no difference. When there is a permit rule evaluating to *I(P)* and a deny rule evaluating to *I(D)*. *Deny-overrides* and *Permit-overrides* will result in the same decision *I(DP)*. Thus, $d(P, q) = d(P', q)$.

For illustration purposes, Table III shows the combined decisions of *Deny-overrides* and *Permit-overrides* applied to one permit rule r_i and one deny rule r_j. *Deny-overrides* and *Permit-overrides* result in different decisions only for the three shaded entries: $d(r_i, q) = Permit \wedge d(r_j, q) = Deny$, $d(r_i, q) = Permit \wedge d(r_j, q) = I(D)$, $d(r_i, q) = I(P) \wedge d(r_j, q) = Deny$. They are corresponding to the conditions in *Theorem 2*.

TABLE III. *DENY-OVERRIDES*/*PERMIT-OVERRIDES* FOR PERMIT/DENY RULES

	$d(r_i, q)=Permit$	$d(r_i, q)=I(P)$	$d(r_i, q)=N/A$
$d(r_j, q)=Deny$	Deny/Permit	Deny/I(P)	Deny
$d(r_j, q)=I(D)$	I(D)/Permit	I(DP)	I(D)
$d(r_j, q)=N/A$	Permit	I(P)	N/A

The following are two examples with both permit and deny rules. *Deny-Overrides* and *Permit-Overrides* are not equivalent in the first example, but equivalent in the second example.

Example 1. *Deny-overrides* and *Permit-overrides* are not functionally equivalent with respect to the following rule list:

r_1: <, age>=18, Permit>,
r_2: <_, sex = female, Permit>,
r_3: <_, age<18, Deny>,
r_4: <_, sex=male, Deny>.

For q= {age=18, sex=male}, $d(r_1, q)=Permit$, $d(r_2, q)=N/A$, $d(r_3, q) = N/A$, and $d(r_4, q)=Deny$. Thus, $d(P, q)=Deny$, $d(P', q)=Permit$, and $d(P, q) \neq d(P', q)$. If the correct response to query q is not *Deny,* then *Deny-overrides* in P must have been used incorrectly. Furthermore, if the correct response is *Permit*, the correct combining algorithm is possibly *Permit-overrides* (it also can be another combining algorithm that produces *Permit*).

Example 2. *Deny-overrides* and *Permit-overrides* are functionally equivalent with respect to the following rules:

r_1: <_, age>=18 $\wedge$ sex=female, Permit>,
r_2: <_, age<18 $\vee$ sex=male, Deny>

The conditions in *Theorem 2* cannot be satisfied: $d(r_1, q) = Permit$ if and only if $d(r_2, q)=N/A$; $d(r_1, q) = N/A$ if and only if $d(r_2, q)=Deny$; if $d(r_1, q) = I(P)$ if and only if $d(r_2, q)=I(D)$.

B. Deny-overrides vs Deny-unless-permit

We classify policies into three categories: (1) R has no default rule. In this case, each rule can evaluate to *N/A, I(D),* or *I(P)*; (2) R contains a default deny rule <_,_,*Deny*>; and (3) R has no default deny rule, but has a default permit rule <_,_, *Permit*>.

Theorem 3. Given policy P=<*PT, Deny-overrides, R*> and P'=<*PT, Deny-unless-permit, R*>. If R has no default rule, then P and P' are not functionally equivalent.

Proof:

(1) The theorem holds for one rule, i.e., R= $\{r_1\}$. Because R does not contain <_,_, *Deny*> or <_,_, *Permit*>, there exists q such that $d(r_1, q) \in \{N/A, I(D), I(P)\}$. Then $d(P, q) = d(r_1, q) \neq Deny$. However, $d(P', q)= Deny$. So $d(P, q) \neq d(P', q)$.

(2) The theorem holds for two rules, i.e., R= $\{r_1, r_2\}$. There exists q such that $d(r_1, q) \in \{I(P), I(D), N/A\}$ and $d(r_2, q) \in \{I(P), I(D), N/A\}$. The following table presents the decisions of combining $d(r_1, q)$ and $d(r_2, q)$ by *Deny-overrides*. None of the entries is *Deny* or *Permit*. However, $d(P', q) =Deny$. Thus, $d(P, q) \neq d(P', q)$.

TABLE IV. *DENY-OVERRIDES* FOR TWO RULES

	$d(r_1, q) =I(P)$	$d(r_1, q) =I(D)$	$d(r_1, q) =N/A$
$d(r_2, q)=I(P)$	I(P)	I(DP)	I(P)
$d(r_2, q)=I(D)$	I(DP)	I(D)	I(D)
$d(r_2, q)=N/A$	I(P)	I(D)	N/A

(3) Suppose the theorem holds for *n-1* rules. $d(P_{n-1}, q) \neq d(P'_{n-1}, q)$, where P_{n-1} = <*PT, Deny-Overrides,* $\{r_1, ..., r_{n-1}\}$ > and P'_{n-1} = <*PT, Deny-unless-permit,* $\{r_1, ..., r_{n-1}\}$>. We show that the theorem also holds for n rules. Note that for any q, $d(P_{n-1}, q)$ is neither *Deny* or *Permit.* In other words, $d(P_{n-1}, q) \in \{ I(P), I(D), I(DP), N/A\}$, and $d(r_n, q) \in \{ I(P), I(D), N/A\}$. Similar to (2), $d(P, q)$ is neither *Deny* or *Permit* as shown in the following table. However, $d(P', q) =Deny$. So $d(P, q) \neq d(P', q)$.

TABLE V. *DENY-OVERRIDES* FOR MORE THAN TWO RULES

$d(r_n, q)$	$d(P'_{n-1}, q)$ =I(P)	$d(P'_{n-1}, q)$ =I(D)	$d(P'_{n-1}, q)$ =I(DP)	$d(P'_{n-1}, q)$ = N/A
=I(P)	I(P)	I(DP)	I(DP)	I(P)
=I(D)	I(DP)	I(D)	I(DP)	I(D)
=N/A	I(P)	I(D)	I(DP)	N/A

Therefore, Theorem 3 holds.

TABLE VI. *SEMANTIC DIFFERENCES BETWEEN FIVE RULE COMBINING ALGORITHMS IN XACML 3.0*

Deny-overrides vs Permit-overrides	***Theorem 1.*** Given policy P = <PT, Deny-overrides, R> and P'= <PT, Permit-overrides, R>. If r_i ($1 \le i \le n$) are all permit rules or r_i ($1 \le i \le n$) are all deny rules, then P and P' are functionally equivalent.
	Theorem 2. Given *policy* P=<PT, Deny-overrides, R> and P'=<PT, Permit-overrides, R>, where R has at least one permit rule and at least one deny rule. For any q, $d(P, q) \neq d(P', q)$ if and only if there exists permit rule r_i=<rt_i, rc_i, Permit>$\in R$, deny rule r_j =<rt_j, rc_j, Deny> $\in R$, and query q, such that: (a) $d(r_i, q) = Permit \wedge d(r_j, q) \in \{Deny, I(D)\}$ or (b) $d(r_i, q) = I(P) \wedge d(r_j, q) = Deny$.
Deny-overrides vs Deny-unless-permit	***Theorem 3.*** Given policy P=<PT, Deny-overrides, R> and P'=<PT, Deny-unless-permit, R>. If R has no default rule, then P and P' are not functionally equivalent.
	Theorem 4. Given policy P=<PT, Deny-overrides, R>, P'=<PT, Deny-unless-permit, R>, and <_, _, Deny> $\in R$. P and P' are functionally equivalent if and only if R has no permit rule.
	Theorem 5. Given policy P=<PT, Deny-overrides, R>, P'=<PT, Deny-unless-permit, R>, <_, _, Deny >$\notin R$, and <_, _, Permit>$\in R$. P and P' are functionally equivalent if and only if R has no deny rule.
Deny-overrides vs Permit-unless-deny	***Theorem 6.*** Given policy P=<PT, Deny-overrides, R> and P'=<PT, Permit-unless-deny, R>. If R has no default rule, then P and P' are not functionally equivalent.
	Theorem 7. Given policy P=<PT, Deny-overrides, R> and P'=<PT, Permit-unless-deny, R>. If <_, _, Deny> $\in R$. Then $d(P, q) = d(P', q) = Deny$.
	Theorem 8. Given *policy* P=<PT, Deny-overrides, R>, P'=<PT, Permit-unless-deny, R>, <_, _, Deny>$\notin R$, <_, _, Permit> $\in R$. P and P' are functionally equivalent if and only if R has no deny rule.
Deny-overrides vs First-applicable	***Theorem 9.*** Given policy P = <PT, Deny-overrides, R>, P'= <PT, First-applicable, R>, and R contains only permit rules. P and P' are not functionally equivalent if and only if there exist i and j ($0<i<j\le n$) such that $d(r_i, q) = I(P)$, $d(r_j, q) = Permit$, and $d(r_k, q) = N/A$ for any $0<k<i$.
	Theorem 10. Given policy P = <PT, Deny-overrides, R> and P'= <PT, First-applicable, R>, where R has at least one deny rule. For any query q, $d(P, q) \neq d(P', q)$ if and only if there exist i and j ($0<i<j\le n$) such that: (a) $d(r_i, q) \in \{Permit, I(P)\}$, $d(r_j, q) \in \{Deny, I(D)\}$, $d(r_k, q) = N/A$ for any $0<k<i$, OR (b) $d(r_i, q) = I(D)$, $d(r_j, q) \in \{Permit, Deny, I(P)\}$, $d(r_k, q) = N/A$ for any $0<k<i$.
Permit-unless-deny vs Deny-unless-permit	***Theorem 11.*** Given policy P=<PT, Permit-unless-deny, R> and P'=<PT, Deny-unless-permit, R>. If R has no default rule, then P and P' are not functionally equivalent.
	Theorem 12. Given policy P=<PT, Permit-unless-deny, R>, P'=<PT, Deny-unless-permit, R>, and <_, _, Permit> $\in R$. P and P' are functionally equivalent if and only if R has no deny rule.
	Theorem 13. Given policy P=<PT, Permit-unless-deny, R>, P'=<PT, Deny-unless-permit, R>, <_, _, Permit > $\notin R$, and <_, _, Deny> $\in R$. P and P' are functionally equivalent iff R has no permit rule.
First-applicable vs Permit-unless-deny /Deny-unless-permit	***Theorem 14.*** Given policy P=<PT, First-applicable, R>, P'=<PT, CA', R> and $CA' \in$ {*Permit-unless-deny*, *Deny-unless-permit*}. If the first rule in R is neither <_,_, Deny> nor <_,_, Permit>, then P and P' are not functionally equivalent.
Permit-overrides vs Permit-unless-deny	***Theorem 15.*** Given policy P=<PT, Permit-overrides, R> and P'=<PT, Permit-unless-deny, R>. If R has not default rule, then P and P' are not functionally equivalent.
	Theorem 16. Given policy P=<PT, Permit-overrides, R>, P'=<PT, Permit-unless-deny, R>, and <_, _, Permit> $\in R$. P and P' are functionally equivalent if and only if R has no deny rule.
	Theorem 17. Given policy P=<PT, Permit-overrides, R>, P'=<PT, Permit-unless-deny, R>, <_, _, Permit>$\notin R$, and < , _, Deny>$\in R$. P and P' are functionally equivalent iff R has no permit rule.
Permit-overrides vs Deny-unless-permit	***Theorem 18.*** Given *policy* P=<PT, Permit-overrides, R> and P'=<PT, Deny-unless-permit, R>. If R has not default rule, then P and P' are not functionally equivalent.
	Theorem 19. Given policy P=<PT, Permit-overrides, R> and P'=<PT, Deny-unless-permit, R>. If <_, _, Permit> $\in R$, then $d(P, q) = d(P', q) = Permit$.
	Theorem 20. Given policy P=<PT, Permit-overrides, R>, P'=<PT, Deny-unless-permit, R>, <_, _, Permit>$\notin R$, <_, _, Deny> $\in R$. P and P' are functionally equivalent if and only if R has no permit rule.
Permit-overrides vs First-applicable	***Theorem 21.*** Given policy P = <PT, Permit-overrides, R>, P'= <PT, First-applicable, R>, and R contains only deny rules. P and P' are not functionally equivalent if and only if exist i and j ($0<i<j\le n$) such that $d(r_i, q) = I(D)$, $d(r_j, q) = Deny$, and $d(r_k, q) = N/A$ for any $0<k<i$.
	Theorem 22. Given policy P = <PT, Permit-overrides, R> and its mutant P'= <PT, First-applicable, R>, where R has at least one permit rule. For any query q, $d(P, q) \neq d(P', q)$ if and only if there exist i and j ($0<i<j\le n$) such that: (a) $d(r_i, q) \in \{Deny, I(D)\}$, $d(r_j, q) \in \{Permit, I(P)\}$, $d(r_k, q) = N/A$ for any $0<k<i$, OR (b) $d(r_i, q) = I(P)$, $d(r_j, q) \in \{Permit, Deny, I(D)\}$, $d(r_k, q) = N/A$ for any $0<k<i$.

C. Summary

Our main observations from the formalization of semantic differences between the rule combining algorithms (Table VI) are summarized as follows.

First, although different rule combining algorithms in XACML are meant to be different mechanisms for rule composition and conflict resolution, any pair of them studied in this paper can be functionally equivalent with respect to certain rules. Even when they are different for a given policy, the difference can be subtle for XACML practitioners to comprehend. According to the informal descriptions about the rule combining algorithms in the XACML standard specification, for example, an inexperienced user may believe that *Deny-overrides* and *Permit-overrides* are different whenever a policy has both deny and permit rules. This is not true per *Theorem 2* and *Example 2*. The subtle differences between the rule combining algorithms increase the likelihood of incorrect uses in real-world XACML applications.

Second, the techniques commonly used in the theorem proofs are induction and decision tables. While the majority of the proofs are somehow straightforward (e.g., for Theorems 1 and 3), several require careful reasoning (e.g., for Theorem 2). The main challenge of this research is not the proofs per se, but the identification and formulation of the theorems.

Third, due to the symmetry between the terms "permit" and "deny", some pairs are similar, or opposite in terms of the occurrences of permit and deny. For example, the comparison of *Deny-overrides* and *Deny-unless-permit* is similar to that of *Permit-overrides* and *Permit-unless-deny*. Using the theorems for *Deny-overrides* vs *Deny-unless-permit* (i.e., *Theorems 3, 4* and *5* in Table VI), we can obtain similar theorems and proofs for *Permit-overrides* vs *Permit-unless-deny* by flipping between *Permit* and *Deny*, and between *I(D)* and *I(P)* (i.e., *Theorems 15, 16* and *17* in Table VI). The symmetry between permit and deny is also useful for discussions on different policy patterns in the next section.

Fourth, the classifications of rules for comparing combing algorithms usually reflect the underlying meanings of the combining algorithm identities. For example, 'unless' is related to default rules because it indicates how default circumstances are handled. When comparing *X-unless-y* with *X-overrides*, rules are classified according to whether there is a default rule. When comparing *X-unless-y* with *First-applicable*, it is critical to consider whether or not the first rule is a default rule.

IV. Policy Patterns

This section applies the results in Section III to several policy patterns where rules are defined on the same set of attributes. Multiple rule combining algorithms can be functionally equivalent or have very subtle differences. Consider the following example:

Example 3. Attributes are *role* (subject attribute), *op* (action attribute), and *data* (resource attribute). *role*$\in${*patient, receptionist, nurse, physician*}, *operation* $\in$ {*read, write*}, and *data* $\in${*patient-record, appointment*}. The rules are defined for three attributes, *role*, *op*, and *data*, as follows.

r_1: <*role=patient*, *op=read* $\wedge$ *data=patient-record*, *Permit*>
r_2: <*role=receptionist*, *op=write* $\wedge$ *data=appointment*, *Permit*>
…
r_9: <*role=physician*, *op=write* $\wedge$ *data=patient-record*, *Permit*>

Here each query has a value for each attribute. For any query that makes one rule evaluate to *I(D)* or *I(P),* then all other rules also evaluate to *I(D)* or *I(P)*. Now we discuss how rule combining algorithms apply to such rules.

- **Permit Rules without Default**

In this pattern, *R* consists of only permit rules that are defined over the same attributes (e.g., rules in *Example 3*). According to *Theorem 2*, *Permit-overrides* and *Deny-overrides* are functionally equivalent. According to *Theorem 9* in Appendix A, *Deny-overrides* and *First-applicable* are also functionally equivalent. This is because there does not exist two permit rules such that one evaluates to *I(P)* and the other evaluates to *Permit* – as mentioned before, if one permit rule evaluates to *I(P)*, then all the other permit rules evaluate to *I(P)*. Similarly, *Permit-overrides* and *First-applicable* are functionally equivalent. In this pattern, *Permit-overrides*, *Deny-overrides*, and *First-applicable* are interchangeable. The combined decision can be *Permit*, *I(P)*, or *N/A*. Obviously, *Deny-unless-permit* (or *Permit-unless-deny*) is different because it results in either *Permit* or *Deny*. Likewise, if *R* consists of only deny rules that are defined over the same attributes, *Permit-overrides*, *Deny-overrides*, and *First-applicable* are functionally equivalent.

- **Permit Rules with Default Deny**

In this pattern, *R* has *n-1* permit rules (*n>1*) followed by one default deny rule <*_, _, Deny*>. The rules are defined over the same attributes. An example is the rules in *Example 3* together with <*_, _, Deny*> as the last rule in the policy. According to *Theorem 22* in Appendix A, *First-applicable* and *Permit-overrides* are not functionally equivalent only when all permit rules evaluate to *I(P)*. In this case, *First-applicable* results in *I(P)* and *Permit-overrides* results in *I(DP).* If it is the final decision to the user, they produce the same decision *Indeterminate*. Therefore, from user's perspective, they are interchangeable. According to *Theorem 14* in Appendix A, *First-applicable* and *Deny-unless-permit* are not functionally equivalent when the permit rules evaluate to *I(P)*. In this case, *Deny-unless-permit* results in *Deny*. For any non-error query (i.e., each permit rule evaluates to *Permit* or *N/A*), however, *First-applicable*, *Permit-overrides,* and *Deny-unless-permit* have the same effect, i.e., *Permit* if one of the permit rules evaluates to *Permit*, and *Deny* if all permit rules evaluate to *N/A*. The only difference among these combining algorithms is how errors are handled. Depending on the application, this difference can be resolved in the policy enforcement point (PEP), which is beyond the scope of this paper.

- **Deny Rules with Default Permit**

In this pattern, *R* has *n-1* deny rules (*n>1*) followed by one default permit rule <*_, _, Deny*>. It is similar to the pattern of "permit rules with default deny" except that deny and permit are flipped. *First-applicable* and *Deny-overrides* are not equivalent only when all deny rules evaluate to *I(D)*. In this

case, *First-applicable* results in *I(D)* and *Deny-overrides* results in *I(DP)*. If it is the final decision to the user, they have the same decision *Indeterminate*. *First-applicable* and *Permit-unless-deny* are not functionally equivalent when the deny rules evaluate to *I(D)*. In this case, *Permit-unless-deny* results in *Permit*. For any non-error query (i.e., each deny rule evaluates to *Deny* or *N/A*), however, *First-applicable*, *Deny-overrides*, and *Permit-unless-Deny* have the same effect. The only difference is how errors are handled.

V. COMPARING POLICY COMBINING ALGORITHMS

In XACML 3.0, the names of rule combining algorithms are all used as policy combining algorithms with similar meanings. The additional policy combining algorithm is *Only-one-applicable*. So this research considers six policy combining algorithms. Policy combining algorithms are used within policy sets. A policy set consists of a target, a policy combining algorithm, and a list of policies and/or policy sets. The given policy combining algorithm combines decisions of policies and other policy sets. For clarity, we focus on policy sets that are composed of policies. Let *PS=<PST, PCA, PL>* be a policy set, consisting of policy set target *PST*, policy combining algorithm *PCA*, and a list of policies *PL*. We revise the problem definition in Section II as follows: Given policy *PS=<PST, PCA, PL>* and a list of policy combining algorithms, determine whether or not *PCA* is used correctly, and if *PCA* is used incorrectly, which policy combining algorithm should be chosen.

Let *PS′=<PST, PCA′, PL>* be a mutant of *PS* with a different policy combining algorithm *PCA′*, and *d(PS, q)* be *PS*'s response to query *q*. We adapt our approach to policy combining algorithms by the following two extensions:

- A main difference between rule combining and policy combining is that policy's decision set ranges from *{Permit, Deny}* (e.g., when the policy's rule combining algorithm is *Deny-unless-permit*) to *{Permit, Deny, N/A, I(D), I(P), I(DP)}*, whereas rule's decision set is *{Permit, N/A, I(P)}* or *{Deny, N/A, I(D)}*. In other words, policy combination algorithms deal with more combinations of decisions. As such, the necessary and sufficient conditions of *Theorem 2* in Section III and other theorems in Appendix A need to be adjusted.
- The algorithms for query generation aim to find attributes and values to make individual rules evaluate to a target decision. For policy combining, individual policies need to be evaluated to their target decisions. This requires reasoning about each policy's target, rules, and rule combining algorithm.

Again, let us take *Deny-overrides vs Permit-overrides* as an example. We first adapt *Theorem 1* to policy combining.

Policy combining version of Theorem 1 (i.e., Theorem 23 in Appendix). Given policy *PS = <PST, Deny-overrides, PL>* and *PS′= <PST, Permit-overrides, PL>*. If all $P_i \in PL$ ($1 \le i \le n$) have only permit (or deny) rules and none of the rule combining algorithms is *Deny-unless-permit* (or *Permit-unless-deny*), then *PS* and *PS′* are functionally equivalent.

The proof is also similar to that of *Theorem 1* because $d(P_i, q) \in \{Permit, I(P), N/A\}$. Here the policies follow the policy patterns in Section III. The above theorem does not work when the rule combining algorithm in an all-permit-rules policy is *Deny-unless-permit* because it may result in *Deny*. This can be handled in the following enhanced version of *Theorem 2*:

Policy combining version of Theorem 2 (i.e., Theorem 24 in Appendix). Given policy set *PS=<PST, Deny-overrides, PL>* and *PS′=<PST, Permit-overrides, PL>,* where *PL* has policies that can evaluate to *Permit* and *Deny*, respectively. For any *q*, $d(PS, q) \neq d(PS', q)$ if and only if there exists policy $P_i \in PL$ and $P_j \in PL$ ($i \neq j$) such that

(a) $d(P_i, q) = Permit \wedge d(P_j, q) \in \{Deny, I(D)\}$ or

(b) $d(P_i, q) = I(P) \wedge d(P_j, q) = Deny$.

The differences between the six policy combining algorithms (i.e., 15 pairs) are formalized by 27 theorems in the appendix. The detailed proofs can be found in [15]. The theorems show that although different policy combining algorithms are meant to be different mechanisms for policy composition, any pair of policy combining algorithms can be functionally equivalent with respect to certain policies.

VI. CASE STUDIES

The main contribution of this research is the formalization of the semantic differences between combining algorithms. It is demonstrated through rigorous proofs. Nevertheless, we have applied the research to nine case studies with different levels of complexity. The case studies are summarized in Table VII. K-market is a sample application of Balana with a total of 12 rules in three policies. It is the only one that is originally encoded in XACML 3.0. *itrust*, *pluto*, *conference*, and *fedora* are real-world policies from literature. They were originally encoded in XACML 2.0 or 1.0. We manually converted them into XACML 3.0 with the same semantics. *itrustX* (*X=5, 10, 20*, or *40*) is a policy synthesized from *itrust*. It has *X* times as many rules as *itrust*. The new rules in *itrustX* are created by replicating the existing rules with new attribute values. Because the real-world policies from literature have a small number of rules, we use *itrustX* to evaluate whether or not our approach is applicable to large-scale policies.

TABLE VII. RESULTS OF CASE STUDIES

Name	***#Rules***	***Combining algorithm***	***Equivalent combining algorithms***
K-market [1]	12	Deny-overrides	None
itrust[2]	64	First-applicable	Permit-overrides/ Deny-overrides
pluto	21	Permit-overrides	None
conference [5]	15	Permit-overrides	None
fedora[3]	12	Deny-overrides	None
itrust5	320	First-applicable	Permit-overrides/ Deny-overrides
itrust10	640	First-applicable	Permit-overrides/ Deny-overrides

[2] http://agile.csc.ncsu.edu/iTrust/wiki/doku.php?id=start
[3] http://www.fedora.info

itrust20	1,280	First-applicable	Permit-overrides/ Deny-overrides
itrust40	2,560	First-applicable	Permit-overrides/ Deny-overrides

Our case studies apply the relevant theorems to each subject policy and determine which combining algorithms are functionally equivalent with respect to the given policy target and rules. The results are shown in Table VII. For *itrust* and its variations, *Permit-overrides* and *Deny-overrides* are equivalent. They follow the policy pattern in Section IV. Note that the equivalence relationships may no longer exist if rules are modified or added to break the underlying assumption of the pattern. For each of the other policies, no equivalent combining algorithms are found.

VII. RELATED WORK

This research is a unique effort in formalizing the semantic differences between combining algorithms in XACML. It is not comparable to the existing literature. It is different from the work on policy composition or combining algorithms (e.g., [10][13]). The work is somehow relevant to verification of XACML policies. The existing methods for verifying XACML policies require user to specify properties (e.g., relationships between access control elements) and then explore the input space to verify the given properties. They do not target incorrect uses of combining algorithms.

Margrave [5] is a model-checking tool for verifying whether an XACML policy satisfies given properties by exhaustively exploring the input space of attributes. Properties describe the constraints on attributes such as roles, resources, and actions. Margrave also provides a tool for change-impact analysis, which summarizes the differences between evolving policies. Focusing on a restricted subset of XACML, Margrave has a very limited support for combining algorithms. As a special application of change-impact analysis in Margrave, Hwang et al. developed a static analysis framework for detecting multiple-duty-related security leakage [8]. Martin et al. have presented a mutation-based verification approach for assessing the quality of properties specified for a policy [11]. It verifies a set of properties against the original policy as well as each of its mutants.

Hughes and Bultan have considered property verification of XACML policies as Boolean Satisfiability problems [7]. They transform XACML policies into a mathematical model and define properties as partial orderings between access control policies. The partial ordering relations specify that a certain type of outcome for one policy subsumes the same type of outcome for another policy. Some of the XACML policy combining algorithms are slightly simplified.

VIII. CONCLUSIONS

We have presented the formalization of the semantic differences between the main combining algorithms in XACML 3.0. It identifies the necessary and sufficient condition of functional equivalence between rule (or policy) combining algorithms with respect to the rules in a given policy (or the policies in the given policy set). The formalization provides an in-depth understanding about the combining algorithms and reduces potential incorrect uses. It is also useful for fault-based testing of combining algorithms in XACML policies.

ACKNOWLEDGMENT

This work was supported in part by US National Science Foundation (NSF) under grant CNS 1359590.

REFERENCES

[1] Balana, "Open source XACML 3.0 implementation," http://xacmlinfo.org/2012/08/16/balana-the-open-source-xacml-3-0-implementation/, 2012.

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

[3] A. Bertolino, S. Daoudagh, F. Lonetti, and E. Marchetti. "Xacmut: Xacml 2.0 mutants generator." Sixth IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), 2013, pp.28-33.

[4] P. Bonatti, S.De.Capitani di Vimercati, and P.Samarati. "An algebra for composing access control policies." ACM Transactions on Information and System Security (TISSEC) 5.1 (2002), pp.1-35.

[5] K. Fisler, S. Krishnamurthi, L.A. Meyerovich and M.C. Tschantz. "Verification and change-impact analysis of access-control policies." Proceedings of the 27th International Conference on Software Engineering. ACM, 2005, pp.196-205.

[6] V.C. Hu, D. Ferraiolo, R. Kuhn, A. Schnizer, K. Sandlin, R. Miller and K. Scarfone. "Guide to Attribute Based Access Control (ABAC) Definition and Considerations." NIST Special Publication 800 (2014): 162.

[7] G. Hughes and T. Bultan. "Automated verification of access control policies using a SAT solver." International Journal on Software Tools for Technology Transfer 10.6 (2008), pp.503-520.

[8] J.H. Hwang, T. Xie, and V.C. Hu. "Detection of Multiple-Duty-Related Security Leakage in Access Control Policies." Third IEEE International Conference on. Secure Software Integration and Reliability Improvement, 2009. pp.65-74.

[9] N. Li, J.C. Mitchell, and W.H. Winsborough. "Design of a role-based trust-management framework." Proc. of the IEEE Symposium on Security and Privacy, 2002, pp.114-130.

[10] D. Lin, P. Rao, and E. Bertino. "Policy decomposition for collaborative access control." Proceedings of the 13th ACM Symposium on Access Control Models and Technologies. ACM, 2008, pp.103-112.

[11] E. Martin, J.H. Hwang, T. Xie, and V. C. Hu. "Assessing quality of policy properties in verification of access control policies." Annual Computer Security Applications Conference, IEEE, 2008, pp.163-172.

[12] E. Martin, and T. Xie. "A fault model and mutation testing of access control policies." Proceedings of the 16th International Conference on World Wide Web. ACM, 2007, pp.667-676.

[13] P. Mazzoleni, B. Crispo, S. Sivasubramanian, and E. Bertino. "XACML policy integration algorithms." ACM Transactions on Information and System Security (TISSEC) 11.1 (2008): 4.

[14] OASIS, "eXtensible Access Control Markup Language (XACML) Version 3.0," http://www.oasisopen.org/committees/xacml/. 2013.

[15] D. Xu, Y. Zhang, N. Shen. "Formalizing semantic differences of combining algorithms in XACML 3.0," Technical Report, Boise State University, http://cs.boisestate.edu/~dxu/research/TR-BSU-CS-SEAL2014-001.pdf

APPENDIX. Semantic Differences between Six Policy Combining Algorithms in XACML3.0

Deny-overrides vs Permit- overrides	***Theorem 23.*** Given policy set *PS = <PST, Deny-overrides, PL>* and *PS′= <PST, Permit-overrides, PL>*. If all $P_i \in PL$ ($1 \le i \le n$) have only permit (or deny) rules and none of the rule combining algorithms is *Deny-unless-permit* (or *Permit-unless-deny*), then *PS* and *PS′* are functionally equivalent.
	Theorem 24. Given policy set *PS=<PST, Deny-overrides, PL>* and *PS′=<PST, Permit-overrides, PL>,* where *PL* has policies that can evaluate to *Permit* and *Deny*, respectively. For any *q*, $d(PS, q) \neq d(PS', q)$ if and only if there exists policy $P_i \in PL$ and $P_j \in PL$ ($i \neq j$) such that: (a) $d(P_i, q) = Permit \wedge d(P_j, q) \in \{Deny, I(D)\}$, or (b) $d(P_i, q) = I(P) \wedge d(P_j, q) = Deny$.
Deny-overrides vs Deny-unless-permit	***Theorem 25.*** Given policy set *PS=<PST, Deny-overrides, PL>* and *PS′=<PST, Deny-unless-permit, PL>*. If there exists *q* such that $d(P, q) \in \{N/A, I(D), I(P), I(DP)\}$ for all $P \in PL$, then $d(PS, q) \neq d(PS', q)$.
	Theorem 26. Given policy set *PS=<PST, Deny-overrides, PL>* and *PS′=<PST, Deny-unless-permit, PL>*. For any *q* such that there exists $P \in PL$ and $d(P, q)=Deny$. $d(PS, q) \neq d(PS', q)$ if and only if there exists $P' \in PL$, $d(P', q)=Permit$.
	Theorem 27. Given policy set *PS=<PST, Deny-overrides, PL>* and *PS′=<PST, Deny-unless-permit, PL>*. For any *q* such that there exists $P \in PL$ and $d(P, q)=Permit$. $d(PS, q) \neq d(PS', q)$ if and only if there exists $P' \in PL$, $d(P', q) \in \{Deny, I(D), I(DP)\}$.
Deny-overrides vs Permit-unless-deny	***Theorem 28.*** Given policy set *PS=<PST, Deny-overrides, PL>* and *PS′=<PST, Permit-unless-deny, PL>*. If there exists *q* such that $d(P, q) \in \{N/A, I(D), I(P), I(DP)\}$ for all $P \in PL$, then $d(PS, q) \neq d(PS', q)$.
	Theorem 29. Given policy set *PS=<PST, Deny-overrides, PL>* and *PS′=<PST, Permit-unless-deny, PL>*. For any *q* such that there exists $P \in PL$ and $d(P, q)=Deny$. $d(PS, q) = d(PS', q)=Deny$.
	Theorem 30. Given policy set *PS=<PST, Deny-overrides, PL>* and *PS′=<PST, Permit-unless-deny, PL>*. For any *q* such that there exists $P \in PL$ and $d(P, q)=Permit$. $d(PS, q) = d(PS', q)$ if and only if there not exists $P' \in PL$, $d(P', q) \in \{Deny, I(D), I(DP)\}$.
Deny-overrides vs First-applicable	***Theorem 31.*** Given policy set *PS=<PST, Deny-overrides, PL>* and *PS′=<PST, First-applicable, PL>*. If there exists *q* such that $d(P, q) \in \{I(P), Permit, I(DP)\}$ for all $P \in PL$, then $d(PS, q) \neq d(PS', q)$ if and only if there exist *i* and j ($0<i<j \le n$) such that (a) $d(P_i, q) = I(P)$, $d(P_j, q) \in \{Permit, I(DP)\}$ and $d(P_k, q) = N/A$ for any $0<k<i$. OR (b) $d(P_i, q) = Permit$, $d(P_j, q) = I(DP)$ and $d(P_k, q) = N/A$ for any $0<k<i$.
	Theorem 32. Given policy set *PS=<PST, Deny-overrides, PL>* and *PS′=<PST, First-applicable, PL>*. If there exists any *q* such that $d(P, q) \in \{ Deny, I(DP), I(D)\}$ where *PL* has at least one *deny* policy. For any query *q*, $d(PS, q) \neq d(PS', q)$ if and only if there exist *i* and j ($0<i<j \le n$) such that: (a) $d(P_i, q) \in \{Permit, I(P)\}$, $d(P_j, q) \in \{Deny, I(DP), I(D)\}$, $d(r_k, q) = N/A$ for any $0<k<i$, OR (b) $d(r_i, q) = I(D)$, $d(r_j, q) \in \{ Deny, I(DP)\}$, $d(r_k, q) = N/A$ for any $0<k<i$. OR (c) $d(r_i, q) = I(DP)$, $d(r_j, q)=Deny$, $d(r_k, q) = N/A$ for any $0<k<i$. (d) $d(r_i, q) = I(P)$, $d(r_j, q)=Permit$, $d(r_k, q) = N/A$ for any $0<k<i$.
Permit-unless-deny vs Deny-unless-permit	***Theorem 33.*** Given policy set *PS=<PST, Permit-unless-deny, PL>* and *PS′=<PST, Deny-unless-permit, PL>*. If there exists any *q* such that $P \in PL$, $d(P, q)$ =Permit, and $P' \in PL$, $d(P', q)$ =Deny, then $d(PS, q) \neq d(PS', q)$.
	Theorem 34. Given policy set *PS=<PST, Permit-unless-deny, PL>* and *PS′=<PST, Deny-unless-permit, PL>*. If there exists any *q* such that $P \in PL$, $d(P, q)$ =Permit, then $d(PS, q) = d(PS', q)$ if and only if there not exists $P' \in PL$, $d(P', q)=Deny$.
	Theorem 35. Given policy set *PS=<PST, Permit-unless-deny, PL>* and *PS′=<PST, Deny-unless-permit, PL>*. If there exists any *q* such that $P \in PL$, $d(P, q)$ =Deny, then $d(PS, q) = d(PS', q)$ if and only if there not exists $P' \in PL$, $d(P', q)=Permit$.
First-applicable vs Permit-unless-deny / Deny-unless-permit	***Theorem 36.*** Given policy set *PS=<PST, First-applicable, PL>*, *PS′=<PST, PCA′, PL>* and $CA' \in \{Permit\text{-}unless\text{-}deny, Deny\text{-}unless\text{-}permit\}$. If there exists any *q* such that P_1 is the first policy in *PL,* such that $d(P_1, q) \in \{ I(DP), I(D), I(P)\}$, then $d(PS, q) \neq d(PS', q)$.

Permit-overrides vs Permit-unless-deny	***Theorem 37.*** Given policy set *PS=<PST, Permit-overrides, PL>* and *PS′=<PST, Permit-unless-deny, PL>*. If there exists *q* such that *d(P, q)* ∈ {*N/A, I(D), I(P), I(DP)*} for all *P*∈*PL*, then *d(PS, q)* ≠ *d(PS′, q)*.
	Theorem 38. Given policy set *PS=<PST, Permit-overrides, PL>* and *PS′=<PST, Permit-unless-deny, PL>*. For any *q* such that there exists *P* ∈*PL* and *d(P, q)=Permit. d(PS, q) = d(PS′, q)* if and only if there not exists *P′* ∈*PL, d(P′, q)= Deny*.
	Theorem 39. Given policy set *PS=<PST, Permit-overrides, PL>* and *PS′=<PST, Permit-unless-Deny, PL>*. For ny *q* such that there exists *P* ∈*PL* and *d(P, q)=Deny*. *d(PS, q) = d(PS′, q)* if and only if there not exists *P′* ∈*PL, d(P′, q)={Permit, I(P), I(DP)}*.
Permit-overrides vs Deny-unless-permit	***Theorem 40.*** Given policyset *PS=<PST, Permit-overrides, PL>* and *PS′=<PST, Deny-unless-permit, PL>*. If there exists *q* such that *d(P, q)* ∈ {*N/A, I(D), I(P), I(DP)*} for all *P*∈*PL*, then *d(PS, q)* ≠ *d(PS′, q)*.
	Theorem 41. Given policyset *PS=<PST, Permit-overrides, PL>* and *PS′=<PST, Deny-unless-permit, PL>*. For any *q* such that there exists *P* ∈*PL* and *d(P, q)=Permit. Then d(PS, q) = d(PS′, q)=Permit*.
	Theorem 42. Given policyset *PS=<PST, Permit-overrides, PL>* and *PS′=<PST, Deny-unless-permit, PL>*. For any *q* such that there exists *P* ∈*PL* and *d(P, q)=Deny*. *d(PS, q) = d(PS′, q)* if and only if there not exists *P′* ∈*PL, d(P′, q)={Permit, I(P), I(DP)}*.
Permit-overrides vs First-applicable	***Theorem 43.*** Given policyset *PS=<PST, Permit-overrides, PL>* and *PS′=<PST, First-applicable, PL>*. If there exists *q* such that *d(P, q)* ∈ {*I(D), Deny, I(DP)*} for all *P*∈*PL*, then *d(PS, q)* ≠ *d(PS′, q)* if and only if there exist *i* and *j (0<i<j≤n)* such that (a) $d(P_i, q) = I(D)$, $d(P_j, q) \in \{Deny, I(DP)\}$ and $d(P_k, q) = N/A$ for any $0<k<i$. OR (b) $d(P_i, q) = Deny$, $d(P_j, q) = I(DP)$ and $d(P_k, q) = N/A$ for any $0<k<i$.
	Theorem 44. Given policyset *PS=<PST, Permit-overrides, PL>* and *PS′=<PST, First-applicable, PL>*. If there exists any *q* such that *d(P, q)* ∈ { *Permit, I(DP),I(P)*} where *PL* has at least one *permit* policy. For any query *q*, *d(PS, q)* ≠ *d(PS′, q)* if and only if there exist *i* and *j (0<i<j≤n)* such that: (a) $d(P_i, q) \in \{Deny, I(D)\}$, $d(P_j, q) \in \{Permit, I(DP), I(P)\}$, $d(r_k, q) = N/A$ for any $0<k<i$, OR (b) $d(r_i, q) = I(P)$, $d(r_j, q) \in \{ Permit, I(DP)\}$, $d(r_k, q) = N/A$ for any $0<k<i$. OR (c) $d(r_i, q) = I(DP)$, $d(r_j, q)=Permit$, $d(r_k, q) = N/A$ for any $0<k<i$. OR (d) $d(r_i, q) = I(D)$, $d(r_j, q)=Deny$, $d(r_k, q) = N/A$ for any $0<k<i$.
Only-one-applicable VS First-applicable	***Theorem 45.*** Given policyset *PS=<PST, Only-one-applicable, PL>* and *PS′=<PST, First-applicable, PL>*. If there exist *i* such that $d(P_i, q) = \in \{Deny, Permit\}$, then *d(PS, q)* ≠ *d(PS′, q)* if and only if there exist *k* and *j (0<k<i<j≤n)* such that $d(P_k, q) = N/A$ for any $0<k<i$, and there exists j>i such that $d(P_j, q) \neq NA$.
Only-one-applicable VS Permit-override	***Theorem 46.*** Given policyset *PS=<PST, Only-one-applicable, PL>* and *PS′=<PST, Permit-override, PL>*. If there exist *i* such that $d(P_i, q) \in \{Deny, Permit, N/A\}$, then *d(PS, q) = d(PS′, q)* if and only if all the other policies are evaluate to *N/A,* such that $d(P_k, q) = N/A$ for any $k \neq i$.
Only-one-applicable VS Deny-override	***Theorem 47.*** Given policyset *PS=<PST, Only-one-applicable, PL>* and *PS′=<PST, Deny-override, PL>*. If there exist *i* such that $d(P_i, q) \in \{Deny, Permit, N/A\}$, then *d(PS, q) = d(PS′, q)* if and only if all the other policies are evaluate to *N/A,* such that $d(P_k, q) = N/A$ for any $k \neq i$.
Only-one-applicable VS Deny-Unless-Permit	***Theorem 48.*** Given policy set *PS=<PST, Only-one-applicable, PL>* and *PS′=<PST, Deny-Unless-Permit, PL>*. If there exist *i* such that $d(P_i, q) \in \{Deny, Permit\}$, then *d(PS, q) = d(PS′, q)* if and only if all the other policies are evaluate to *N/A,* such that $d(P_k, q) = N/A$ for any $k \neq i$.
Only-one-applicable VS Permit-Unless-Deny	***Theorem 49.*** Given policy set *PS=<PST, Only-one-applicable, PL>* and *PS′=<PST, Permit-Unless-Deny, PL>*. If there exist *i* such that $d(P_i, q) \in \{Deny, Permit\}$, then *d(PS, q) = d(PS′, q)* if and only if all the other policies are evaluate to *N/A,* such that $d(P_k, q) = N/A$ for any $k \neq i$.