A Verified Algorithm for Detecting Conflicts in XACML Access Control Rules

Michel St-Martin Amy P. Felty

School of Electrical Engineering and Computer Science, University of Ottawa, Canada
mstma016@uottawa.ca, afelty@eecs.uottawa.ca

Abstract

We describe the formalization of a correctness proof for a conflict detection algorithm for XACML (eXtensible Access Control Markup Language). XACML is a standardized declarative access control policy language that is increasingly used in industry. In practice it is common for rule sets to grow large, and contain unintended errors, often due to conflicting rules. A conflict occurs in a policy when one rule permits a request and another denies that same request. Such errors can lead to serious risks involving both allowing access to an unauthorized user as well as denying access to someone who needs it. Removing conflicts is thus an important aspect of debugging policies, and the use of a verified algorithm provides the highest assurance in a domain where security is important. In this paper, we focus on several complex XACML constructs, including time ranges and integer intervals, as well as ways to combine any number of functions using the boolean operators *and*, *or*, and *not*. The latter are the most complex, and add significant expressive power to the language. We propose an algorithm to find conflicts and then use the Coq Proof Assistant to prove the algorithm correct. We develop a library of tactics to help automate the proof.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifiying and Verifying and Reasoning about Programs; K.6.5 [*Management of Computing and Information Systems*]: Security and Protection

Keywords program correctness, formal verification, access control, policy analysis, Coq, XACML

1. Introduction

XACML (eXtensible Access Control Markup Language) [11] is a policy specification language that allows policies to be defined in a wide variety of domains. It is an OASIS [10] standard that is becoming more widely used, especially in recent years (see e.g., [12]). Its expressive power allows access control policies to strike a balance between secure and flexible access; policies must prevent access when there is a security risk and allow access when required, such as in a medical emergency when a doctor needs immediate access to medical records. Errors in policies can pose serious risks, and conflicts in policy rules are a common source of errors.

We present an algorithm for detecting conflicts in policies, and implement it and prove its correctness in the Coq Proof Assistant [1, 4]. We do not cover all of XACML here, but do cover a significant sublanguage of XACML 3.0 [11]. The main restriction is that we do not cover all of the XACML's defined *functions*. Policies express various conditions on *attributes* and their *values*, and a request is composed of pairs of attributes and their values. Functions are used in a rule to express how components of a request *match* components of the rule, in order to determine which requests the rule applies to. For example, suppose a rule allows access to a particular student to enter the Formal Methods Research Lab (FMLab) during its opening hours. The function `integer-equals` may be used to indicate that the student's ID number must match the subject attribute in a request in order for the rule to apply. Other allowable functions include those that can test membership in a bag, for instance, testing if the resource attribute in the request is one of `Lab1`,. . .,`LabN`. Still other examples include functions for testing whether or not the time attribute in a request (representing the time the request was made) matches the time constraints expressed in the rule.

This work reports on and significantly extends the work in the first author's thesis [14] and our workshop paper [15], which in turn extend previous work on detecting conflicts in Cisco firewall rules [2]. The format of firewall rules is fairly restricted and rules express constraints on a fixed number of attributes in a fixed order. XACML is a significantly more general and expressive language, even considering the sublanguage considered here, mainly due to allowing a much larger set of attributes as well as fairly complex conditions on these attributes. As a result, conflict detection is considerably more complex than for firewall rules. The statement of correctness and its proof are not complicated, but require considering a large number of cases, including many "corner cases" that can be difficult to get right. In fact, in an earlier version of this proof, we stated a hypothesis to help simplify the proof development. When we later went back to remove this hypothesis, we found it was inconsistent, and a few of these corner cases relied on this inconsistency. Fortunately, these errors were not difficult to fix, but they did require a slight modification to the affected cases in the conflict detection program.

In order to handle the added complexity, the work involved some effort in automating proofs using Coq's Ltac facility [4]. This automation helped to both simplify the proofs and shorten the proof text. Even so, at about 4000 lines, the proof script is significantly larger than the one in [2]. We were able to reuse a small part of the previous proof development because the high-level statement of correctness and its proof are the same here. The underlying lemmas and definitions that these proofs depend on however, are significantly different.

CPP'16, January 18–19, 2016, St. Petersburg, FL, USA
© 2016 ACM. 978-1-4503-4127-1/16/01...$15.00
http://dx.doi.org/10.1145/2854065.2854079

Many policy languages, including XACML, have their own built-in rule-combining algorithms for resolving conflicts. XACML's choices include "deny-overrides," "permit-overrides," "first-applicable," and "only-one-applicable." The first two allow a policy writer to explicitly specify that a rule that denies (respectively, permits) access always overrides a rule that permits (respectively, denies) access. The third specifies that the first rule in sequential order that applies to a request is the one that takes priority. The last states that if more than one rule or policy applies, no decision will be returned. Policy writers make use of these rule-combining mechanisms, but unintended errors in the form of conflicts are still common. Many policies, for instance, are developed over a period of time, often with a change in personnel, where one policy administrator develops the initial policy, and later administrators make changes. The conflict detection algorithm we present here provides a tool for static analysis and policy debugging. It identifies all conflicts. Policy maintainers decide which ones are errors that need fixing, and which ones can remain and be handled by the rule-combining algorithm. For example, a user taking advantage of "first-applicable" may ignore conflicts arising from rules intended as exceptions that are placed before default rules.

In the next section (Sect. 2), we present the encoding of policies and requests in Coq using a running example. In Sect. 3, we present the conflict detection algorithm, and in Sect. 4, we discuss its proof of correctness. We discuss future work in Sect. 5, related work in Sect. 6, and we conclude in Sect. 7.

In the Coq code presented below, `Prop` is Coq's built-in type of propositions, while `bool` is in `Set` and is Coq's boolean datatype; the latter is used in the conflict detection program. We use Coq's built-in lists and integers. The `::` operator is an infix cons operator on lists, `++` is infix append, and `In` is used for list membership. We use both Coq's comparison operators on integers, such as `=`, `<`, `<=`, etc., and their boolean counterparts, whose names end in "`b`". For example, `(i<j)` is in `Prop`, while `(i <b j)` is in `bool`. The connectives `/\`, `\/`, and `~` are used to construct propositions, while `&&`, `||`, and `negb` are the corresponding boolean operators. In addition to these, we also have a type called `trilean` which is like a boolean but with an extra non-applicable value. It is only used in cases where the policy file contains unexpected values or functions. In general, they can be thought of as booleans. They use three values, `T`, `F`, and `NA`, which correspond to true, false, and non-applicable values, respectively. They include the usual connectives, denoted as `/\t`, `\/t`, and `~t`. They work the same way as the usual boolean operators, with `NA` taking priority in conjunction and disjunction (e.g. `T /\t NA = NA`). The files of the Coq formalization are available at `www.eecs.uottawa.ca/~afelty/cpp16/`.

2. Encoding XACML Policies in Coq

We present the part of XACML that involves specifying individual rules within a policy, and we present only enough of the XACML syntax (which uses XML) to enable us to describe the components of a rule and how they are encoded in Coq. A rule is made up of a *target*, an optional set of *conditions*, and an *effect* that indicates whether the rule will *permit* or *deny* access. A target groups together the *subject*, *resource*, and *action* components of a rule. In the example mentioned earlier, the student's ID number is the subject, "enter" is the action, and FMLab is the resource. The lab's opening hours are expressed in the rule's condition. Figure 1 illustrates how this rule could be expressed in XACML. We have elided many parts for readability, for example name `integer-equal` is short for `urn:oasis:names:tc:xacml:1.0:function:integer-equal`, and we only show the general format excluding resources, actions, and conditions, whose syntax is similar syntax to subjects.

Multiple targets are allowed, simply by adding more elements of the form `<AllOf>...</AllOf>` in the `Target`'s section. For a

```
<Rule RuleId="Rule3" Effect="Permit">
  <Target> <AnyOf> <AllOf>
    <Match MatchId="integer-equal">
    <AttributeValue DataType="integer">
       123</AttributeValue>
    <AttributeDesignator
       AttributeId="student_ID"
       Catagory="access-subject".../>
    </Match>
    ...
  </AllOf> </AnyOf> </Target>
  <Condition>...</Condition>
</Rule>
```

Figure 1. Example XACML rule

rule to apply, any of the targets mentioned must match a request. Conditions are similar to targets except that they must all be satisfied for a rule to apply, as opposed to needing only one target. If the target doesn't include any subject in its definition, then the rule applies regardless of which subject made a request. Similarly, if the resources, action or even the entire target is missing, then a request always applies to the resource, action or target (respectively) portion of the rule. Note that one aspect of XACML that we ignore is the use of global targets, i.e., those that are found at the policy-level and apply to every rule in the policy file. Here we have only considered local rule-level targets. This omission is without loss of generality, since the information in a global target can be instead included in the target of each individual rule, resulting in an equivalent (but possibly more verbose) policy.

We illustrate the encoding of XACML rules in Coq via an example. Consider a policy for students to access computing and research laboratories at a university. We assume there are undergraduate and graduate computing laboratories where students work on their course work, and some number of research laboratories that each have graduate students assigned to them. We initially include the following rules: (1) anyone is allowed (undergraduates, graduate students, and professors) to enter the undergraduate computing laboratory during its opening hours, (2) graduate students and professors are allowed to enter the graduate computing laboratory during its opening hours, (3) the student with ID number 123 is allowed to enter FMLab during its opening hours, and (4) the student with ID number 456 is allowed to enter the AI Research Lab during its opening hours. We give the Coq definition of a rule first and then fill in the details.

```
Inductive effect: Set := permit | deny.
Record rule: Set :=
  ruleCons {eff: effect; rules: srac}.
```

A rule is a Coq record with two fields. The first field has type `effect`, defined as an enumerated type with two elements via a Coq inductive definition. The second field combines four of XACML's elements and is of type `srac`, which stands for "subject-resource-action-condition." In XACML, all of these elements use the same representation and can be combined to form complex rules using logical operators, as defined below.

```
Inductive srac : Set :=
| single : core -> nat -> srac
| and : srac -> srac -> srac
| or : srac -> srac -> srac
| not : srac -> srac.
```

In `single`, the first argument, of type `core`, is used to encode the core XACML functions such as "integer-equal" shown in Figure 1. These will be explained in more detail later. Its second argument

is a natural number. These numbers indicate which field from the requests the rule uses. XACML would reference the field by name, and allow them in any order, but we chose to map them to specific numbers for easier access. For this example, the mapping of field names to numbers is 1: ID number, 2: location, 3: action, 4: current time, 5: registration status and 6: subject type. Note that the fields follow this order for this example only but the list can be of any size and any order. The constructors `and`, `or`, and `not` are used to combine `single` elements of type `srac` in the obvious way.

We present Coq code for rules (2) and (3) above before filling in the definition of `core` and other missing details.

```
Definition Rule2 := ruleCons permit
  (and (and (and (or Grads Professors)
    GradLab) Enter) GlabHours).
Definition Rule3 := ruleCons permit
  (and (and (and G123 FMLab) Enter)
    FMlabHours).
```

In the encoding of rules, `Grads`, `Professors`, `GradLab`, `Enter`, `GlabHours`, `G123`, `FMLab`, and `FMlabHours`, are all elements of type `srac` (whose definitions we discuss later). Note that `Rule3` is an equivalent rule to the XACML coded rule in Figure 1. Their effects are both `permit`. The XACML function "student ID = 123" from the figure is translated into `G123`. The parts omitted from the XACML code translate to the resource `FMLab`, the action `Enter`, and the condition `FMlabHours`.

In order to fill in the missing definitions, we first need to define the `core` type, which is the type of the first field of the `single` constructor in the `srac` type. Its definition (below) shows how we encode XACML's main functions in Coq:

```
Definition MIDNIGHT := 86400.
Definition Z' := {z:Z | 0 <= z <= MIDNIGHT}.
Definition Z'NM := {z:Z | 0 <= z < MIDNIGHT}.

Inductive core : Set :=
| any : reqValue -> core
| empty : reqValue -> core
| intInRange : Z -> Z -> core
| intGt : Z ->  core
| intLt : Z -> core
| timeInRange : Z' -> Z' -> core
| na : core.
```

There are four main constructors implementing these functions, which have arguments of types `Z` and `Z'`. `Z` is Coq's built-in integer type, and `Z'` encodes the subset of the integers that represent time, restricting time values to be between 0 and the number of seconds in 24 hours. Our time ranges include their lower bound but exclude the upper bound. In particular, when we write (`timeInRange` m M), a time t in a request is considered to match if $m \leq t < M$, i.e., t is in the interval $[m, M)$. The type `Z'NM` is similar but excludes midnight. We will see later that it is used in requests. Note that a request includes the time it was made as one of the attributes, and if a request is done at midnight, it will never meet any of the time range conditions within rules because no time range starts at the upper bound. Because of this requests can't take the value `MIDNIGHT`, otherwise it would cause problems with correctness, which is why requests are from `Z'NM` instead of `Z'`.

The constructor `intInRange` is used to encode XACML integer ranges. With ranges, we can also represent integer equalities by setting both ends of the range to the same value. Similarly, `intGt` and `IntLt`, in combination with integer equality, are used to encode not only *greater-than* and *less-than* functions, but also *greater-than-or-equal* and *less-than-or-equal*. We combine these functions to reduce the number of functions needed to be implemented without losing any power of expression. By reducing the number of implemented functions, we reduce code size and increase efficiency. We especially gain efficiency when performing conflict detections. This algorithm must compare fields of two rules, and minimizing the number of functions greatly reduces the numbers of pairs of functions that need to be compared. If we had included the three extra functions mentioned, the number of pairs to check would increase from $7^2 = 49$ (because of the 7 constructors) to $10^2 = 100$. Note that the core functions were chosen in a way that makes them a closed set under conjunction and negation (negating sometimes leads to multiple core functions combined with `srac`'s `or` constructor).

The first constructor, `any`, represents that any subject, resource, action, or condition (depending on its location) applies as long as the field has the specified type, while the second, `empty`, is for when no value applies. The `na` constructor is used in error cases which only occur when the policy file uses functions we haven't yet implemented, or uses unexpected values such as a decimal when an integer is expected. The `reqValue` inductive type appears in the types of `any` and `empty`. Its main use is to denote types and values for requests. In practice policies are used to evaluate requests, but it is convenient to express some of the elements of policy rules using some of the constructors of `reqValue`. Its definition is:

```
Inductive reqValue: Set :=
| timeReq : Z'NM -> reqValue
| intReq : Z -> reqValue
| blank : reqValue.
```

The values of the fields in a request (in our example, these can be any of the 6 attribute/value pairs listed above) can either be an integer representing a valid time (`timeReq`), an integer encoding any of the other fields such as resources and subjects (`intReq`), or `blank` when a particular field is absent from a particular request. We use `Z'NM` here because we do not want the value of `MIDNIGHT` occuring in requests for the reason mentioned above.

We can now fill in the definitions of the particular fields used in our encoding of rules (2) and (3). Most of the XACML functions occurring in `Rule2` and `Rule3` are strings encoded as integers, represented in Coq as follows:

```
Definition intEq (z:Z)(n:nat):srac :=
  single (intInRange z z) n.

Definition Grads: srac := intEq 1 6.
Definition Professors: srac := intEq 2 6.
Definition GradLab := intEq 2 2.
Definition Enter: srac := intEq 0 3.
Definition G123: srac := intEq 123 1.
```

In the definition of `intEq`, the first argument is an integer value or the integer encoding of a string, while the second argument is the index of the field it is referencing. For example, `Grads` and `Professors` represent two different subject types (field number 6) and `G123` represents the student whose student number is 123 (field number 1).

The only other XACML functions occurring in this example are those in the definitions of `GlabHours` and `FMlabHours`, which use `timeInRange`, one of the constructors in the definition of `core`, and `not`, the last constructor in the definition of `srac`. They are represented as follows:

```
Lemma ThreeAm_rc : 0 <= 10800 <= MIDNIGHT.
Definition ThreeAm:Z' :=
  (exist _ 10800 ThreeAm_rc).

Definition GlabHours: srac :=
  not (single (timeInRange ThreeAm FourAm) 4).
Definition FMlabHours:srac :=
```

```
  single (timeInRange SixAm ElevenPm) 4.
```

The types `Z'` and `Z'NM` were defined earlier but not yet used in examples. Elements of both of these dependent types are a pair containing an integer and a proof that the integer is in the restricted range. The above example illustrates this representation for 3am. Other times are represented similarly.

We assume for this example that students can access the graduate lab anytime except for between 3am and 4am, and that the formal methods lab is open from 6am to 11pm. We note that in XACML, the time range `GlabHours` would simply be expressed as the range 4am to 3am which would "wrap around" midnight, with 3am on the next day. We separate these ranges into two sections. The reasoning behind this will be explained later.

To continue our example, suppose that the university has decided to update the lab access policy, instituting a new rule that (5) denies access to research labs to students whose registration has lapsed. Suppose also that (6) students who have violated some university policy are no longer allowed to enter research labs after 5pm. In this example, we will assume that student 123 has violated some rule. We add the following two rules (and associated definitions).

```
Definition NotRegistered := intEq 0 5.
Definition Violations := intEq 123 1.
Definition After5:srac :=
  single (timeInRange FivePm Z'MIDNIGHT) 4.
Definition AnyTime: srac :=
  single (any (timeReq Z'NM0)) 4.
Definition AnyLocation :=
  single (any (intReq 0)) 2.
Definition Rule5 := ruleCons deny
  (and (and (and NotRegistered
    (or FMLab AILab)) Enter) AnyTime).
Definition Rule6 := ruleCons deny
  (and (and (and Violations AnyLocation)
    Enter) After5).
```

For any `z` in the range `0 <= z <= MIDNIGHT`, we write `Z'z` to denote its corresponding value in `Z'` (and omit its definition here), and similarly for any `z` in the range `0 <= z < MIDNIGHT`, we write `Z'MNz`. Note that the definition of `NotRegistered` is having value `0` for registration status (field number 5), and that the definition of `Violations` includes only student 123. Given these definitions, `Rule5` and `Rule6` directly encode the new rules. Note that these rules introduce some conflicts. A particular graduate student who is not registered may be both allowed and denied access to one of the research labs. To avoid this conflict, all rules permitting access must be restricted to registered students only. This first kind of conflict is a result of being denied and permitted access to the same resource. The other kind of conflict introduced here is due to intersecting time ranges. Since the resource component of rule (6) is `AnyLocation`, this rule applies to all labs. There is a conflict with rule (3), for instance, since student 123 is allowed and denied access to FMLab during the hours between 5pm and 11pm.

Our conflict detection program reports the following list of pairs of rules to be in conflict: (1,6)::(2,6)::(3,5)::(3,6)::(4,5)::nil.[1] While this example is simple, it is quite common for new rules to be added that don't take into account all the necessary content of the old rules, causing bugs requiring the update of existing rules to fix them.

As mentioned earlier, a policy is a set of rules, and policies are used to evaluate requests. A policy is represented here using Coq lists (since we reference rules by their placement in the list). To illustrate requests, suppose the registered (registration status 1) graduate (subject type 1) student 123 wants to enter (action 0) the graduate lab (location 2) at 3pm. The encoding in Coq of this specific example is as follows:

```
Definition Request1 :=
  (intReq 123)::(intReq 2)::
  (intReq 0)::(timeReq ThreePm)::(intReq 1)::
  (intReq 1)::nil.
```

Note that there are exactly six elements in this list, corresponding to the six attribute/value pairs, and that they appear in order by field number.[2]

3. Detecting Conflicts in Policy Rules

Consider the conflict mentioned above between rules (3) and (6): student 123 is allowed and denied access to the graduate lab during the hours between 5pm and 11pm. There is a conflict because all fields of both rules "overlap". The subjects overlap since rule 3 applies to student 123 and rule 6 applies to any student with violations (such as student 123). The resource (FMLab) in rule (3) is one of several labs covered by rule (6). The action fields in both rules are exactly the same, and finally, there is overlap in the two time ranges in the conditions of these rules (the first covers 6am-11pm, and the second covers 5pm to midnight). The key to the algorithm is correctly defining this overlap. Figure 2 defines the main function `coreCheck`, which detects overlap between two elements of type `core`, then returns this overlap (as a `core`). This function must consider every possible case for each of the two arguments. Note, that there is never any overlap between constructors whose arguments have different types (e.g. `intLt` and `timeInRange` never overlap), which reduces the number of cases from 49 to 21. We remind the reader that we currently have 2 types—integers and time.

The function in Figure 2 starts by eliminating any cases with different types (using the function `typeDiff`) and those that contain errors such as ranges having an upper bound that is smaller than its lower bound (using the function `invalidArgs`). Note that we return `na` instead of `empty` when either `invalidArgs` or `typeDiff` is true because if we negate that `core`, it should still report the error as opposed to returning `any` (the negation of empty). The function then goes through each remaining pair of `core` functions to see how each intersect. The first pairs involve `any` and `empty`. If `c1` is the function `any`, regardless of what `c2` is, the overlap between `any` and `c2` will be `c2`, which is returned. Similarly, if `c1` is the function `empty`, then no request value can apply to its function, thus no request can apply to both `c1` and `c2` thus `empty` (=`c1`) is returned. Also, if `c2` is either `any` or `empty` then `c1` or `empty`, respectively, is returned.

The next example we consider is when both `c1` and `c2` are time ranges. The time range $[m_1, M_1)$ (i.e., any t such that $m_1 \leq t < M_1$) and the time range $[m_2, M_2)$ overlap if $m_1 < M_2$ and $m_2 < M_1$. If this is the case, then they intersect at the time range $[max(m_1, m_2), min(M_1, M_2))$, which is returned by the function. If either of $m_1 < M_2$ and $m_2 < M_1$ are false, there is no overlap, and thus `empty (timeReq Z'NM0)`. Note that `any` and `empty` have arguments (of type `reqValue`), which denote their type. We arbitrarily chose the values `timeReq Z'NM0` and `intReq 0` to be used as time and integer types, but any other values can be used.

In [15] we had originally used "wrapping" time ranges, just as XACML does. Such a time range was denoted by having the upper bound smaller than the lower bound of the time range. By allowing this type of time range, instead of the single `if` statement in the `(timeInRange m2 M2)` case of the `coreCheck` function, which has two conditions (expressing inequalities such as `(m1 <b M2)`),

[1] Actually, our algorithm begins rule numbering at 0, so what we have called `Rule1` is denoted as 0. We have added 1 to all numbers in the output presented here for readability.

[2] In particular, recall that the arguments of the requests for this example are ordered as follows :1: ID number, 2: location, 3: action, 4: current time, 5: registration status, and 6: subject type.

```
Definition coreCheck (c1 c2: core) : core  :=
if typeDiff (type c1) (type c2) then na else
if invalidArgs c1 then na else
if invalidArgs c2 then na else
match c1 with
| any _ => c2
| empty _ => c1

| timeInRange m1 M1 =>
  match c2 with
  | any (timeReq z2) => c1
  | empty (timeReq z2) => empty (timeReq Z'NM0)
  | timeInRange m2 M2 => if m1 <b M2 /\b m2 <b M1
     then timeInRange (max' m1 m2) (min' M1 M2)
     else empty (timeReq Z'NM0)
  | _ => na
  end

| intGt m1 =>
  match c2 with
  | any (intReq z2) => c1
  | empty (intReq z2) => empty (intReq 0)
  | intInRange m2 M2 => if m1 <b M2
     then intInRange (max (m1+1) m2) M2
     else empty (intReq 0)
  | intGt m2 => intGt (max m1 m2)
  | intLt M2 => if m1 <b (M2-1)
     then intInRange (m1+1) (M2-1)
     else empty (intReq 0)
  | _ => na
  end
...
end.
```

Figure 2. Detecting overlap in rule components

```
Definition notCheck (c : core) (n:nat) : srac :=
if invalidArgs c then single na n else
match c with
| any r => single (empty r) n
| empty r => single (any r) n
| timeInRange m M => ...
| intInRange m M => or
    (single (intLt m) n) (single (intGt M) n)
| intGt m => single (intLt (m+1)) n
| intLt M => single (intGt (M-1)) n
| na => single na n
end.

Function removeNots (s:srac)
  {measure notSize s} : srac :=
match s with
| not (single c1 n) => notCheck c1 n
| not (or s1 s2) => and (removeNots (not s1))
                        (removeNots (not s2))
| not (and s1 s2) => or (removeNots (not s1))
                        (removeNots (not s2))
| not (not s1) => removeNots s1
| and s1 s2 => and (removeNots s1)
                   (removeNots s2)
| or s1 s2 => or (removeNots s1)
                 (removeNots s2)
| single c n => s
end.
```

Figure 3. Removing `not` from rules

we needed to express four different cases with nine conditions total. By using XACML's complexity to our advantage (now that we have implemented the `or` operator), we can split these time ranges into two time ranges: the first one starting at time 0 and ending at the upper bound of the original, and the second starting at its lower bound and ending at midnight. Rejoining them with `or` essentially treats them as separate rules. By splitting them up into two separate ranges, we also avoid the problem of having two distinct overlaps. For example, if one time range starts at 5am and ends at 4am, and the other time range starts at 3am and ends at 6am, the overlap is described as two separate time ranges: between 3am and 4am and between 5am and 6am. By splitting the first time range into two ranges, we still get both overlaps, but by treating them as separate rules, we simplify the process of finding them, which increases efficiency.

For the cases when `c1` is `(intGt m1)`, first consider the simple case when `c2` has the same form: `(intGt m2)`. There is always an overlap, where the overlap is `(intGt (max m1 m2))`. In the case when `c2` is `(intLt M2)`, then there is overlap if and only if $m_1 < (M_2 - 1)$ with the overlap being the range $[(m_1 + 1), (M_2 - 1)]$.

We previously said that `core` (using `or`) was built so that it was closed under `not`. We can in fact transform the rules into an equivalent set that does not contain `not`, which will reduce the complexity of our conflict detection algorithm. We do this with the `removeNots` function in Figure 3, which calls `notCheck`. An abbreviated version of the latter function also appears in the figure. The `notCheck` function works by changing each case of `core` into accepting every value it previously wouldn't, rejecting the ones it previously would, and removing the negation. For example, The opposite effect of `(intGt m)` is the function `(intLt (m+1))`. Ranges are slightly more complicated as the unaccepted range is two separate intervals. We can, however, use the `or` constructor to handle this situation. For example, the negation of `(single (intInRange m M) n)` is `(or (single (intLt m) n) (single (intGt M) n))`. Time ranges act similarly to integer ranges except with special cases for when its bounds contain either 0 or midnight. Note that this function can only be used when the `srac` is in negation normal form, that is when negations are only negating atoms. To do this, we can use DeMorgan's laws. The `removeNots` function performs this operation by pushing all occurrences of `not` inward, calling `notCheck` when it gets to a leaf.

We also need to convert the `srac` into disjunctive normal form. We do so with the code in Figure 4. The function `convertDNF` is used by the code in Figure 5, which performs a kind of "normalization" that transforms the tree structure that comes from the logical connectives to a list of lists. The inner lists denote a conjunction of elements of type `srac`, and the outer list represents a disjunction of its elements. First `toLists` is called, which converts one rule's `srac` (named `s`) to normal form with all occurrences of `not` removed as described above (see Figure 3). Having rules in DNF is important because it allows us to treat each disjunct as its own rule. After converting to DNF, `split` will separate the disjuncts. Once it gets to an `and`, because we are in DNF with no negation, we know that all that will be found below will be conjuncts. We can pass control to `split'` which uses a modified merge sort (see Coq manual [4] for a similar merge sort), that uses the field's identifying numbers as sorting keys to combine all of the rule's parts. We modified the case when the two rules use the same key; we combine them using `coreCheck` in Figure 2 instead of adding both entries to the sorted list.

```
Fixpoint distr' (s1 s2 : srac) :=
match s2 with
|or s3 s4 => or (distr' s1 s3) (distr' s1 s4)
| _ => and s1 s2
end.

Fixpoint distr (s1 s2 : srac) : srac :=
match s1 with
| or s3 s4 => or (distr s3 s2) (distr s4 s2)
| _ => distr' s1 s2
end.

Fixpoint DNF (s : srac) : srac :=
match s with
| or s1 s2 => or (DNF s1) (DNF s2)
| and s1 s2 => distr (DNF s1) (DNF s2)
| _ => s
end.

Definition convertDNF (s : srac) : srac :=
  DNF (removeNots s).
```

Figure 4. Converting rules to disjunctive normal form

```
Fixpoint merge l1 l2 :=
let fix merge_aux  l2 :=
match l1, l2 with
| nil, _ => l2
| _, nil => l1
| (c1, n1)::l1', (c2, n2)::l2' =>
    if blt_nat n1 n2
       then (c1, n1) :: merge l1' l2
    else if beq_nat n1 n2
       then (coreCheck c1 c2, n1) ::
            merge l1' l2'
    else (c2, n2) :: merge_aux l2'
end
in merge_aux l2.

Fixpoint split' (s : srac) :
  list (core * nat) :=
match s with
| and s1 s2 => merge (split' s1) (split' s2)
| single c n => (c, n) :: nil
| _ => nil
end.

Fixpoint split (s : srac) :
  list (list (core * nat)) :=
match s with
| or s1 s2 => split s1 ++ split s2
| _ => split' s :: nil
end.

Definition toLists (s :srac) :
  list (list (core*nat)) :=
  split (convertDNF s).
```

Figure 5. Normalizing rules

```
Inductive trilean : Set :=
| T : trilean | F : trilean| NA : trilean.

Fixpoint nonEmptyCore (c :core) : trilean :=
if invalidArgs c then NA else
match c with
| empty _ => F
| na => NA
| _ => T
end.

Fixpoint nonEmptyRule
  (lc : list (core*nat)) : trilean :=
match lc with
| nil => T
| (c,n):: lc' =>
  nonEmptyCore c /\t nonEmptyRule lc'
end.

Fixpoint lCheck (l1 : list (list (core*nat)))
  (l2 : list (core*nat)) : trilean :=
match l1 with
| nil => F
| l :: l1' =>
  nonEmptyRule (merge l l2 \/t lCheck l1' l2)
end.

Fixpoint llCheck (l1 l2 :
  list (list (core*nat))) : trilean :=
match l2 with
| nil => F
| l :: l2' => lCheck l1 l \/t llCheck l1 l2'
end.
```

Figure 6. Finding overlap

The next step is to compare different rules using the code in Figure 6. The `llCheck` function takes 2 rules' `srac` components that have been converted into their list of lists equivalent, and then recursively takes each sub-list from the first argument and each sub-list from the second argument and uses the `merge` function from Figure 5 on each pair. If any of the merged rule parts are non-empty, the rules overlap.

The top-level code for detecting conflicts appears in Figure 7. If two rules have overlap and they also have a different effect, the rules are in conflict. Note that we could use the same approach to find redundancies by simply changing the code to look for the same effect instead of different ones.

To detect if two rules conflict, we go through the policy, represented as a list of rules, to find all conflicts. This part of the implementation comes directly from [2], and we do not repeat it here.[3] Figure 7 shows the type of the main function called `find_conflicts` (which uses two other helper functions not shown). Together, they define a recursive traversal of the list of rules. For each rule r in the list, all rules occurring after r are tested for conflict with r by calling `conflict_check`. This function returns true when the effects from each rule are different (tested by `effectDiff`), the rules overlap (tested by `llCheck`), and neither rule contain errors (tested by `validCores`, not shown). The `find_conflicts` function returns a list of pairs of indices of those rules that conflict.

[3] The only modification is that the call to the new `conflict_check` function defined here instead of the one in [2].

```
Definition effectDiff:
  effect -> effect -> bool := fun a1 a2 =>
match a1, a2 with
| permit, deny => true
| deny, permit  => true
| _, _ => false
end.

Definition trileanTrue (t:trilean) : bool :=
match t with
| T => true
| _ => false
end.

Definition conflict_check: rule ->
 rule -> bool := fun r1 r2 =>
match r1 with (ruleCons e1 s1) =>
  match r2 with (ruleCons e2 s2) =>
    effectDiff e1 e2 && trileanTrue
      (llCheck (toLists s1) (toLists s2)) &&
      validCores (toLists s1) &&
      validCores (toLists s2)
  end
end.

Definition find_conflicts :
  list rule -> list (nat*nat).
```

Figure 7. Finding all conflicts

4. Correctness

In order to express correctness of our algorithm, we must first define declaratively what it means for two rules to conflict, and then show that our implementation finds exactly those pairs of rules that satisfy this definition. As discussed earlier, a conflict occurs between two rules if one rule permits a request and another denies that same request, expressed directly as follows:

```
Definition rule_conflict (r1 r2: rule): Prop:=
  exists rq:list reqValue,
  (rule_permit rq r1 /\ rule_deny rq r2) \/
  (rule_deny rq r1 /\ rule_permit rq r2).
```

To fill in this definition, we must define `rule_permit` (defined in Figure 8) and `rule_deny` (omitted since it is similar). The `rule_permit` function takes a request and a rule, checks to see if the rule's effect is `permit`, and checks that the request applies to the rule using function `sracMatch`. This function evaluates each branch recursively until it gets to a `core` function, which tests (in `coreMatch`) whether the request's field applies to the particular function from the rule. The `sracMatch` function then combines the evaluations using rules for `and`, `or` and `not`.

We also define the following (omitting its definition and just showing its type):

```
Definition rs_conflict:
  list rule -> nat -> nat -> Prop := ...
```

This property is defined using `rule_conflict` and holds when the two rules at the given indices in the list of rules are in conflict. Both `rs_conflicts` and `rule_conflict` are the same as in [2], though the definitions they rely on (e.g., `rule_permit`, `sracMatch`, etc.) are, of course, quite different.

The correctness of the algorithm is expressed by the two theorems at the bottom of Figure 9. Recall that the function call `(find_conflicts rs)` (discussed in Sect. 3 and defined in Figure 7) returns a list of integer pairs, corresponding to the index of each rule, which the algorithm finds to be in conflict. Thus, the proposition `(In (i,j) (find_conflicts rs))` should be true if and only if there exists a request which applies to both rules as defined by `(rs_conflict rs i j)`. In the completeness theorem, we add the condition that `(i < j)` simply to avoid finding each conflict twice. These theorems use many lemmas to help prove their claims, and the most important ones appear in the figure. Most of the work and lines of code in the Coq formalization come from the proofs of these lemmas. For example, we proved the correctness of the overlap for each non-trivial pair of `core` functions that we cover.

The lemma named `coreCheckCorrect` is the correctness lemma for every core function, which uses the lemmas for each specific pair to complete its proof. This lemma proves both the soundness and completeness of this part of the algorithm (specifically the `coreCheck` function). We were also required to show that all the conversions we did resulted in equivalent rules. This property is stated as lemma `llCorrect`. After this lemma are the lemmas `llCheckSound` and `llCheckComplete` which go a step

```
Definition coreMatch
  (reqv:reqValue) (c: core) : trilean :=
if invalidArgs c then NA else
match reqv with
|  timeReq req =>
   match c with
   | any (timeReq r) => T
   | empty (timeReq r) => F
   | timeInRange m M =>
       if m <=b req /\b req <b M
       then T else F
   | _ => NA
   end
|  intReq req =>
   match c with
   | any (intReq r) => T
   | empty (intReq r) => F
   | intInRange m M =>
       if m <=b req /\b req <=b M
       then T else F
   | intGt m => if m <b req then T else F
   | intLt M => if req <b M then T else F
   | _ => NA
   end
| blank => NA
end.

Fixpoint sracMatch (reql : list reqValue)
  (s : srac) : trilean :=
match s with
| single c n =>
    coreMatch (nth n reql blank) c
| and s1 s2 =>
    sracMatch reql s1 /\t sracMatch reql s2
| or s1 s2 =>
    sracMatch reql s1 \/t sracMatch reql s2
| not s1 => ~t sracMatch reql s1
end.

Definition rule_permit (rq : list reqValue)
  (rl : rule) : Prop :=
  eff rl = permit /\
  sracMatch rq (rules rl) = T.
```

Figure 8. Determining if a rule applies to a request

```
Lemma coreCheckCorrect: forall (c1 c2 : core) (reqv : reqValue),
  coreMatch reqv (coreCheck c1 c2) = ((coreMatch reqv c1) /\t (coreMatch reqv c2)).

Lemma llCorrect:forall (s : srac) (lr : list reqValue), llMatch (toLists s) lr = sracMatch lr s.

Lemma llCheckSound : forall (s1 s2: srac), llCheck (toLists s1) (toLists s2) = T ->
  validCores (toLists s1) = true -> validCores (toLists s2) = true ->
  exists lr : list reqValue, (llMatch (toLists s1) lr /\t llMatch (toLists s2) lr) = T.

Lemma llCheckComplete : forall (s1 s2: srac), (exists lr : list reqValue,
  (llMatch (toLists s1) lr /\t llMatch (toLists s2) lr) = T) ->
  llCheck (toLists s1) (toLists s2) = T.

Theorem conflict_check_soundness: forall r1 r2: rule,
  conflict_check r1 r2 = true -> rule_conflict r1 r2.

Theorem conflict_check_completeness: forall r1 r2: rule,
  rule_conflict r1 r2 -> (conflict_check r1 r2 = true).

Theorem conflicts_soundness: forall (rs:rule_set)(i j:nat),
  In (i, j) (find_conflicts rs) -> rs_conflict rs i j.

Theorem conflicts_completeness: forall (rs:rule_set)(i j:nat),
  (i < j) -> rs_conflict rs i j -> In (i, j) (find_conflicts rs).
```

Figure 9. Lemmas and theorems for correctness

further and prove the correctness for `llCheck`. Taking one more step further, we find two theorems (`conflict_check_soundness` and `conflict_check_completeness`) stating the correctness for `conflict_check`. These theorems are used to prove the main theorems for correctness for the highest level of the algorithm (`conflicts_soundness` and `conflicts_completeness`). Their proofs follow the same structure as the proofs found in [2].

5. Future Work

We have proven that the XACML functions never introduce `na` unless there were problems at the policy level (lemma not shown). In the current implementation, the tests for correctness happen at each step of the algorithm. Future work will involve removing these tests and having one that tests the entire policy file before running conflict detection. This would improve the efficiency of the conflict detection algorithm, and separate error checking from conflict detection.

A number of XACML's core functions and types remain unimplemented, thus future work will involve adding more of these to the algorithm's code base. We have built the program and proofs with expansions in mind, thus few changes to the current code and proofs should be needed. For example, if we wanted to add a new type and/or new functions, we simply need to add the type to `reqValue`, add the functions to `core`, add how they work in `coreMatch`, how they combine with other core functions in `coreCheck`, and prove the overlaps stated in `coreCheck` are correct. On a similar topic, we use integers to encode string equality functions, but are currently working on adapting our code to use Coq's string library. This library would be a more natural fit, but it is not as mature as the libraries for numbers.

Currently, we can only run our code in Coq after having translated a XACML policy into Coq code. Once more of XACML has been implemented, we will extract the code into OCaml and write a parser (and prove it correct) so that we can test XACML policies in their original XML format. We will then test the code to see how it scales for large policies. As the code is based on our previous work [2], we expect similar positive results for scaling.

Another direction for this work is to improve the library of tactics we developed for proof automation. For example, they could be generalized further to prevent proofs from changing too much in situations such as the one we encountered, where we needed to redo many proofs in the process of fixing an incorrect hypothesis. These tactics could not only help with simplifying our current proofs, but also aid in proving lemmas after the expansion of our results to new functions and types.

The top level of the algorithm uses the algorithm described in [2]. While there is nothing wrong with the correctness or efficiency of this algorithm, we plan to re-implement this part in order to return more useful information. In particular, we will return the overlaps as opposed to simply stating which rules are in conflict. We have designed this algorithm so that it knows what the conflicts are at each step and returns them, but we have not yet incorporated it into the correctness proof. This is because we started by directly adopting the structure of the algorithm in [2].

Our algorithm was specifically designed with XACML in mind, however, since XACML is such a general language, our algorithm should be able to find conflicts between rules written in any other language that defines its rules with an effect and a combination of functions that apply to particular fields.

6. Related Work

There is much work on developing algorithms and tools for analyzing policies. We focus mainly on other work on conflict detection for XACML, though we also mention a few examples of work on conflict detection for other languages as well as other kinds of analyses developed specifically for XACML.

With regard to our own past work, as mentioned, this work extends our work on conflict detection for firewalls [2]. We have also been involved in work on a tool for administering XACML policies, with a focus on usability aspects for policy administrators that do not necessarily have a technical background [16, 17]. An implementation of conflict detection (unverified) was also part of that work. We have also worked on an analysis that compresses

XACML policies with the goals of reducing risks of conflicts and improving the performance of access granting tools [18].

Huonder [8] developed a conflict algorithm that is generic in the sense that in order to work with concrete XACML policies it must first be extended with definitions (of XACML functions and intersections). He also proposes ways to resolve conflicts. For example, he proposes automatically repairing them by replacing the rule set with an equivalent one without conflicts. In order to do so, whenever there is a conflict, it has to be resolved according to the default resolution policy. For example, if "first-applicable" is chosen, and a rule that denies the request comes first in the policy, then all conflicting rules that appear later have to be changed so that they don't cover this request. In other words, the overlap has to be determined and removed. His algorithms are quite different from ours and not verified. Also, this may resolve the conflicts, but does not necessarily remove the bugs, and furthermore an automatically modified policy makes it harder for the policy administrator to read and understand it. In contrast, our approach provides an opportunity for the administrator to examine each conflict and determine the best way to resolve it him/herself.

In [9], a formal tool is used to analyze policies. Policies and requests are modeled in the Alloy analyzer, and first-order queries are presented to answer questions such as whether or not there are two rules in a given policy that conflict. The subset of XACML considered is simpler than ours. For example, conditions are not considered, and thus complications such as those resulting from time constraints are not handled.

In [3], the author proposes another generic conflict algorithm which can be modified to work with any policy language in which policies are formulated as sets of rules. This algorithm needs to be extended with definitions in order to be used with specific languages. Our work goes much deeper into XACML than this paper does, allowing our algorithm to be used in actual policies (although simplified ones at the moment).

In [13] a method for intersecting policy files is proposed. This work is targeted towards companies trying to merge their policies, in order to get a combined policy which includes the rules from each individual company.

In [6], the authors do a change-impact analysis in which they detect the set of requests which have their access effect changed when the rule set is modified. They first convert the policies into a decision-diagram format. When a rule is added to a policy, the algorithm reports the set of requests such that a permit becomes a deny (or a deny a permit). We note that this algorithm could be extended to detect conflicts by removing rules one at a time. Each time a change is detected, the rule removed is in conflict with at least one other rule in the set. The intersection between the removed rule and the conflicting rules is the set of requests the algorithm reported as being changed. A drawback of this approach is that it can't report which rules the removed rule conflicts with. Another drawback is that this algorithm can't be used to find redundant rules. In our algorithm, as mentioned, we can make one small change to detect redundant rules—simply replacing the call to `effectDiff` with a call to a function that checks for both rules having the same effect.

In [7], the authors perform verification of access control policies using a SAT solver. They develop a formal model for access policies that includes combinators that model the policy-combining algorithms of XACML. They are able to prove, for example, that a large policy that may be obtained by combining several smaller policies (possibly with different combinators attached to different parts of the policy) correctly denies or permits various accesses. As the authors point out, the semantics of the rule combining operators are quite complex. They simplify these semantics, use the simplified model for their verification experiments, but also discuss how the original semantics can be captured by their model. This kind of analysis is quite different than ours. We detect conflicts and give the policy maintainer the opportunity to remove them so that policy decisions do not rely on the rule-combining mechanisms to correctly resolve conflicts dynamically. Also, they only consider conditions on integers and booleans. In addition, they restrict their analysis to policies that use bounded domains.

Conflict detection is also considered as part of the work in [5]. The authors illustrate how term rewriting can be used to show that policies satisfy various properties, including consistency, which indicates the lack of conflicts. They show how this property follows from the standard property of confluence, which can be checked by many modern rewriting tools. They also identify a variety of conditions under which policies can be composed and still retain consistency. Their work does not address the question of finding and reporting individual conflicts, but instead focuses on this and other global properties. Also, their examples focus on simple conditions, which don't include time constraints. They also show how the policy-combining operators of XACML can be incorporated as rewrite rules. Their handling of these operators suggest a way in which we could also incorporate them, in our case, by generalizing the definition of conflict. Working out the details of this generalization is another subject of future work.

7. Conclusion

We have presented an algorithm for detecting conflicts in XACML rules for a substantial subset of the XACML policy language that includes fairly complex conditions. The algorithm is relatively simple; in fact, because of the goal of proving it correct, it was simplified over time as the proof proceeded. We have fully verified its correctness in Coq, along the way defining tactics to automate proofs, designed to be general enough to be reused in many of the lemmas in the proof development. Detecting (and removing) conflicts is an important part of the debugging process for access control policies.

Our contributions beyond the work reported in our workshop paper [15] are numerous. Perhaps the most significant was the incorporation of the `and`, `or`, and `not` operators, which greatly extend the expressiveness of the sublanguage of XACML that we now handle. We also now allow more than one condition in rules, which also extends the expressiveness to include an important part of XACML. Another significant improvement is our new approach to handling time ranges, whose benefits were discussed earlier. Finally, we cover more integer functions of XACML and have extended the set of requests that can now be handled. (In our previous work, requests were limited to four arguments, but we have generalized the algorithm and proof to handle requests represented as lists where each element of the list can use the logical operators such as `and` to form compound expressions.)

As mentioned, one of the main directions of future work is to extend our results to cover the full expressive power of XACML. In the context of the work described in [17], a conflict detection program that covers a larger sublanguage of XACML was implemented. The implementation in [17] is in Java and Prolog, and thus the focus in that work was not on designing an algorithm in Coq that could be proved correct. However, establishing correctness was understood to be important from the beginning and we believe that although significant extensions will be required to the current formal proof development to handle this larger subset, there will be no significant obstacles.

Acknowledgments

The authors would like to thank Bernard Stepien for sharing his expertise on XACML, and the anonymous reviewers for helpful comments.

References

[1] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Springer, 2004.

[2] V. Capretta, B. Stepien, A. Felty, and S. Matwin. Formal correctness of conflict detection for firewalls. In *ACM Workshop on Formal Methods in Security Engineering*, pages 22–30. ACM Press, 2007.

[3] J. Chomicki, J. Lobo, and S. Naqvi. A logic programming approach to conflict resolution in policy management. In *7th International Conference on Principles of Knowledge Representation and Reasoning*, 2000.

[4] Coq Development Team. *The Coq Proof Assistant Reference Manual*, 2009. `coq.inria.fr/distrib/V8.4/refman/`.

[5] D. J. Dougherty, C. Kirchner, H. Kirchner, and A. S. de Oliveira. Modular access control via strategic rewriting. In *12th European Symposium On Research In Computer Security (ESORICS)*, volume 4734 of *Lecture Notes in Computer Science*, pages 578–593, 2007.

[6] K. Fisler, S. Krishnamurthi, L. A. Meyerovich, and M. C. Tschantz. Verification and change-impact analysis of access-control policies. In *27th International Conference on Software Engineering*, pages 196–205, 2005.

[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):473–534, 2008.

[8] F. Huonder. Conflict detection and resolution of XACML policies. Master's thesis, University of Applied Sciences Rapperswil, 2010.

[9] M. Mankai and L. Logrippo. Access control policies: Modeling and validation. In *5th NOTERE Conference*, pages 85–91, 2005.

[10] OASIS. eXtensible Access Control Markup Language (XACML) TC. `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml`, 2004.

[11] OASIS. *XACML Version 2.0*, 2004. `docs.oasis-open.org/xacml/2.0/access_control-xacml-2.0-core-spec-os.pdf`.

[12] OASIS. OASIS members demonstrate interoperability of XACML access control standard in HITSP health care scenario. `http://www.oasis-open.org/news/oasis-news-2008-04-07.php`, 2008.

[13] P. Rao, D. Lin, E. Bertino, N. Li, and J. Lobo. An algebra for fine-grained integration of XACML policies. In *14th ACM Symposium on Access Control Models and Technologies*, pages 63–72, 2008.

[14] M. St-Martin. A verified algorithm for detecting conflicts in XACML access control rules. Master's thesis, University of Ottawa, 2011.

[15] M. St-Martin and A. P. Felty. A verified algorithm for detecting conflicts in XACML access control rules. In *Informal Proceedings of the Workshop on Automated Reasoning in Security and Software Verification (ARSEC)*, pages 4–11, 2013. `http://www.site.uottawa.ca/~afelty/dist/arsec13.pdf`.

[16] B. Stepien, A. Felty, and S. Matwin. A non-technical user-oriented display notation for XACML conditions. In *E-Technologies: Innovation in an Open World, 4th International MCETECH Conference*, pages 53–64. Springer LNBIP, 2009.

[17] B. Stepien, S. Matwin, and A. Felty. Strategies for reducing risks of inconsistencies in access control policies. In *5th International Conference on Availability, Reliability, and Security*, pages 140–147. IEEE Computer Society, 2010.

[18] B. Stepien, A. Felty, and S. Matwin. An algorithm for compression of XACML access control policy sets by recursive subsumption. In *7th International Conference on Availability, Reliability, and Security*. IEEE Computer Society, 2012.