Computing Science Group

On the modelling and analysis of Amazon Web Services access policies

David Power
Mark Slaymaker
Andrew Simpson

CS-RR-09-15

Oxford University Computing Laboratory
Wolfson Building, Parks Road, Oxford, OX1 3QD

Abstract. Cloud computing is a conceptual paradigm that is receiving a great deal of interest from a variety of major commercial organisations. By building systems which run within cloud computing infrastructures, problems related to scalability and availability can be reduced, and, from the point of view of consumers of such infrastructures, abstracted away from. As such infrastructures tend to be shared, it is important that access to the sub-components of each system is controlled. One of the first languages for controlling access to services within a cloud is the Amazon Web Services access policy language. In this paper we present two formal models of this language—one in Z and one in Alloy—and show how the Alloy model might be used to test properties of multiple policies and to generate and test candidate policies.

1 Introduction

Cloud computing (see, for example, [1]) is a conceptual paradigm that is receiving a great deal of interest from a variety of major commercial organisations. By building systems which run within cloud computing infrastructures, problems related to scalability and availability can be reduced, and, from the point of view of consumers of such infrastructures, abstracted away from. As such infrastructures tend to be shared, it is important that access to the sub-components of each system is controlled.

Many cloud computing infrastructures have emerged over the past few years; at the time of writing, Amazon Web Services (AWS) [2] is one of the most widely used. AWS consists of a number of different components, which can be used in combination or alone. One common usage model is to use Elastic Compute Cloud (EC2) instances to process information and to use the Simple Queue Service (SQS) [3] to handle requests and responses. For example, a language translation service might involve an end-user initially submitting input to a web page. Then the inputted string would form part of a request placed in a queue (the request queue). The EC2 instance would consume messages from the queue, perform the required task (in this case, translation), and then put a message containing the result in a second queue (the response queue). The result will then be consumed by the web site.

If all of the sub-components of a system use the same security credentials, it is possible to restrict access using an 'all-or-nothing' approach. However, there are situations where more complex controls are appropriate. For example, there may be a key requirement that a particular service is only available during certain time periods. For this reason, the AWS access policy language was introduced, which enables access to be restricted based on a number of factors, including the time of the request and the originating IP address, as well as more common factors, such as the action that is being performed and the resource that is being acted upon. Currently, only the SQS service supports the AWS access policy language, but there are plans for it to be used on additional components.

As the complexity of access control policies increases, there is a corresponding increase in the risk that a mistake might be made when defining these policies.

The value of analysing such policies has been demonstrated by the work of others in the community, such as Ryan and colleagues (see, for example, [4]) and Bryans and Fitzgerald (see, for example, [5])).

In this paper we seek to reduce that risk with the appropriate application of formal methods. We use a hybrid approach of using both the Z specification language [6] and the Alloy modelling language [7]. Each language has its strengths and weaknesses, with Z better suited to formal proof and Alloy being better suited to automatic analysis. The differences in the languages reflect the intentions of their creators with each having its place. In this paper the Z model is used as a starting point for the Alloy model, which we use for the examples presented. From a pragmatic perspective, our choice of leveraging both languages comes down to the fact that there may be some circumstances in which a fully formal proof is necessary; typically, however, the excellent support for model finding offered by Alloy will be appropriate.

In Section 2 we provide a necessarily brief description of the AWS access policy language. In Section 3 we present a model of the AWS policy language written in the Z specification language. In Section 4 we translate (manually) the Z model into the Alloy language and also extend the model to support the specifics of policies written for the SQS service. We give two examples of the use of the Alloy model in Section 5. In the first example we look at a scenario where multiple queues are used as part of a simple system; we then use the Alloy Analyzer to find examples of situations in which the system will fail to perform as required. In the second example we look at the use of the Alloy Analyzer to assist in the creation of policies that meet a set of complex requirements. By creating a set of requests with known outcomes we can generate candidate policies. Finally, in Section 6, we summarise the contribution and explore avenues of potential future work.

2 Background

The AWS access policy language allows one to construct policies that have a tree-like structure, consisting of sub-components, each of which may give an independent result—with these results being combined systematically to arrive at a final decision. In this respect, it has similarities with the OASIS standard XACML (eXtensible Access Control Markup Language) [8].

The AWS access policy language makes permit or deny decisions based on the identity of the user, the action they are trying to perform, and the resource they are trying to act on. Users are identified by a *principal*, which is used as part of the authentication process and is tied to a specific AWS account. In addition, a number of environmental factors may be used as part of the decision making process, these are identified by the use of keys.

Each policy consists of a number of statements which contain a description of the requests they apply to, plus an effect, which may be permit or deny. Each statement contains lists of actions, lists of resources and lists principals, plus a

number of conditions which must be met. The conditions are related to the key values of the request.

If multiple statements match a request, then deny effects take precedence over permit effects. If no statements match, then the effect is referred to as a soft deny: that is to say that final effect will be deny unless another policy has an effect of permit.

Conditions have a type, which is a matching relation such as string equality or 'before' on date-time values. Conditions also contain a number of clauses, all of which must hold for the condition to be met. Each clause consists of a key and a number of values: if any of the values match the request's key value then the clause holds.

When writing a policy for use with the SQS service a number of additional restrictions apply. Each policy may only contain statements relating to a single queue, the identity of which is used as the resource value. Only the following actions may be used: `ReceiveMessage`; `SendMessage`; `DeleteMessage`; `ChangeMessageVisibility`; and `GetQueueAttributes`.

The only available keys are the standard keys, which are: `CurrentTime (DateTime)`; `SecureTransport (Boolean)`; `SourceIP (IP Address)`; and also `UserAgent (String)`.

The available condition types are dependent on the types of the available keys, and include `DateEquals`, `DateLessThan`, `IpAddress`, and `StringEquals`.

In addition, there is also a restriction that all policies and statements are uniquely identified.

The following example (from [3]) illustrates a policy in which all users are given `ReceiveMessage` permission for the queue named `987654321098/queue1`, but only between noon and 3:00 p.m. on January 31, 2009.

```
{
  "Version": "2008-10-17",
  "Id": "Queue1_Policy_UUID",
  "Statement":
     {
        "Sid":"Queue1_AnonymousAccess_ReceiveMessage_TimeLimit",
        "Effect": "Allow",
        "Principal": {
           "AWS": "*"
        },
        "Action": "SQS:ReceiveMessage",
        "Resource": "/987654321098/queue1",
        "Condition" : {
           "DateGreaterThan" : {
              "AWS:CurrentTime":"2009-01-31T12:00Z"
           },
           "DateLessThan" : {
              "AWS:CurrentTime":"2009-01-31T15:00Z"
           }
        }
     }
}
```

One of the benefits of this language is that, when compared to, for example, XACML, it is relatively streamlined: making a mapping to a formal representation more straightforward than would typically be the case.

3 A Z representation

In this section we describe the structure of an AWS access policy using the Z specification language. The model represents all aspects of the policy language, but does not include any explicit value types. To ease readability, some types are used before they are defined.

Each policy has a unique identifier, a policy language version number (which currently has no practical impact upon policies or their evaluation, but is included for the sake of completeness) and a non-empty list of statements. As the order of statements has no effect on request evaluation, the list is represented as a set. There is currently only one version of the policy language, represented by the constant $v1$.

$[PolicyID]$

$Version ::= v1$

Policy

$version : Version$
$id : PolicyID$
$statements : \mathbb{P}_1\ Statement$

Each statement places conditions upon the requests to which it applies: if the statement applies, it may have an effect which can be to either allow or deny access. During evaluation, a deny decision may be be the result of a statement with an effect of deny—called a *hard deny*—or the absence of an applicable statement with an effect of allow—called a *soft deny*. The absence of an effect in a statement is modelled as a soft deny. In our model, we have explicitly captured the soft deny effect; there is no need to model the hard deny effect explicitly.

Similarly to a policy, each statement has a unique identifier. It also details the principals, actions and resources to which it applies. The principals represent the identity of the requester, the actions represent the action to be performed, and the resource represent the entity that the action will be performed on. Each of these can take multiple values and are represented by non-empty sets. Finally, a statement may contain a number of conditions which further restrict the applicability of a statement.

$[StatementID, Principal, Action, Resource]$

$Effect ::= Allow \mid Deny \mid SoftDeny$

Statement

$sid : StatementID$
$effect : Effect$
$principals : \mathbb{P}_1\ Principal$
$actions : \mathbb{P}_1\ Action$
$resources : \mathbb{P}_1\ Resource$
$conditions : \mathbb{P}\ Condition$

Conditions consist of a matching relation, referred to as the type of the condition, and a number of clauses describing the values to be matched. Each clause contains a key and a number of values. Each key represents a specific piece of data about the request, such as the current time or the IP address from which the request originated. These are compared with concrete values using the matching relation corresponding to the type of the condition.

Condition

$type : CondType$
$clauses : \mathbb{P}_1\ Clause$

$[Key, Value]$

Clause

$key : Key$
$values : \mathbb{P}_1\ Value$

CondType

$match : Value \leftrightarrow Value$

As well as having a model of the policy, it is also necessary to model a request. Each request contains a single principal, action and resource. In addition, the *keys* function provide the value associated with each key. It is assumed that the *keys* function is total.

Request

$principal : Principal$
$action : Action$
$resource : Resource$
$keys : Key \rightarrow Value$

At the top level, a request is evaluated against a set of policies: if any policy evaluates to *Deny* or no policy evaluates to *Allow*, then the result is a *Deny*. If there are no *Deny*s and at least one *Allow*, then the result is *Allow*.

$$
\begin{array}{|l}
EvalPolicies : Request \to (\mathbb{P}\, Policy) \to \{Allow, Deny\} \\
\hline
\forall\, r : Request \bullet EvalPolicies(r) = \\
\quad (\lambda\, pols : \mathbb{P}\, Policy \bullet Deny) \\
\quad \oplus \\
\quad (\lambda\, pols : \mathbb{P}\, Policy \mid \\
\qquad (\exists\, p : pols \bullet EvalPolicy(r)(p) = Allow) \bullet Allow) \\
\quad \oplus \\
\quad (\lambda\, pols : \mathbb{P}\, Policy \mid \\
\qquad (\exists\, p : pols \bullet EvalPolicy(r)(p) = Deny) \bullet Deny)
\end{array}
$$

The evaluation of a policy is dependent on the evaluation of the statements it contains. When no statement results in a *Deny* or *Allow*, the result is a *SoftDeny*.

$$
\begin{array}{|l}
EvalPolicy : Request \to Policy \to Effect \\
\hline
\forall\, r : Request \bullet EvalPolicy(r) = \\
\quad (\lambda\, p : Policy \bullet SoftDeny) \\
\quad \oplus \\
\quad (\lambda\, p : Policy \mid \\
\qquad (\exists\, s : p.statements \bullet EvalStatement(r)(s) = Allow) \bullet Allow) \\
\quad \oplus \\
\quad (\lambda\, p : Policy \mid \\
\qquad (\exists\, s : p.statements \bullet EvalStatement(r)(s) = Deny) \bullet Deny)
\end{array}
$$

If a request matches the constraints of a statement, then the evaluation results in the value of the effect attribute, otherwise it evaluates to soft deny. For a request to match the constraints of a statement, the principal, action and resource of the request must each be contained in the corresponding set in the statement; in addition, all of the conditions must be met.

$$
\begin{array}{|l}
EvalStatement : Request \to Statement \to Effect \\
\hline
\forall\, r : Request \bullet EvalStatement(r) = \\
\quad (\lambda\, s : Statement \bullet SoftDeny) \\
\quad \oplus \\
\quad (\lambda\, s : Statement \mid (r, s) \in MatchStatement \bullet s.effect)
\end{array}
$$

$$
\begin{array}{|l}
MatchStatement : Request \leftrightarrow Statement \\
\hline
MatchStatement = \\
\quad \{r : Request;\ s : Statement \mid \\
\qquad r.principal \in s.principals \land \\
\qquad r.action \in s.actions \land \\
\qquad r.resource \in s.resources \land \\
\qquad (\forall\, c : s.conditions \bullet (r, c) \in MeetsCondition)\}
\end{array}
$$

To meet a condition, each of the clauses must be met using the matching relation specified by the type attribute. The key of the clause defines which of

the request keys to use for each clause. For a match to be made, at least one of the clause values must match the key value.

$$
\begin{array}{|l}
MeetsCondition : Request \leftrightarrow Condition \\
\hline
MeetsCondition = \\
\quad \{r : Request;\ c : Condition \mid \\
\qquad (\forall\, cl : c.clauses \bullet \\
\qquad\quad (\exists\, v : cl.values \bullet (r.keys(cl.key), v) \in c.type.match))\}
\end{array}
$$

4 Alloy model

Having presented our Z description of the AWS access policy language, we now consider an Alloy representation, which is, via the Alloy Analyzer, more amenable to automatic analysis. We also describe domain-specific extensions for the SQS service. In Section 5 we will use the extended model both to find example policies and to test the properties of existing policies.

The domain-specific extensions are based around the types of values that can be used in a policy, the matching relations for the values, and the request keys used to access the values. In addition, the actions that can be performed are domain-specific. All of these are entities declared using abstract signatures.

```
abstract sig Value, Key, Action {}
abstract sig CondType {
  match : Value -> Value
}
```

The remaining signatures required for the policies are translated directly from the Z model.

```
sig PolicyId {}
abstract sig Version {}
one sig v1 extends Version {}
sig Policy {
  version : Version,
  pid : PolicyId,
  statements : some Statement
}
sig StatementId, Principal, Resource {}
abstract sig Effect {}
one sig Allow, Deny, SoftDeny extends Effect {}
sig Statement {
  sid : StatementId,
  effect : Effect,
  principals : some Principal,
  actions : some Action,
  resources : some Resource,
  conditions : set Condition
}
sig Condition {
  type : CondType,
  clauses : some Clause
}
sig Clause {
  key : Key,
  values : some Value
}
```

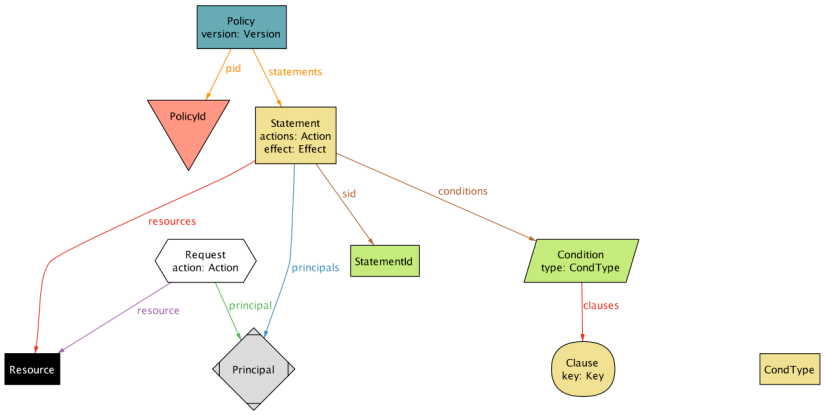

Fig. 1. Policy metamodel

```
sig Request {
  principal : Principal,
  action : Action,
  resource : Resource,
  keys : Key -> one Value
}
```

The metamodel of the basic policy signatures is shown in Figure 1.

The evaluation logic for the Alloy model is also a translation of that of the Z model.

```
fun EvalPolicies ( r : Request, ps : set Policy ) : Effect {
  (some p : one ps | EvalPolicy[r,p] = Deny) => Deny else
  ((some p : one ps | EvalPolicy[r,p] = Allow) => Allow else Deny)
}
fun EvalPolicy ( r : Request, p : Policy ) : Effect {
  (some s : one p.statements | EvalStatement[r,s] = Deny) => Deny else
  ((some s : one p.statements | EvalStatement[r,s] = Allow) => Allow else SoftDeny)
}
fun EvalStatement ( r : Request, s : Statement ) : Effect {
  MatchStatement[r,s] => s.effect else SoftDeny
}
pred MatchStatement ( r : Request, s : Statement ) {
  r.principal in s.principals
  r.action in s.actions
  r.resource in s.resources
  all c : one s.conditions | MeetsCondition[r,c]
}
pred MeetsCondition ( r : Request, c : Condition ) {
  all cl : one c.clauses |
    some v : one cl.values | (r.keys[cl.key] -> v) in c.type.match
}
```

SQS policies support five types of actions (`ReceiveMessage`, etc.—as listed in Section 2). The request keys are: `CurrentTime`, which is a date-time value; `SecureTransport`, which is a Boolean value; `SourceIP`, which is an IP address

value; and `UserAgent`, which is a string value. For strings and IP addresses, it is possible for clauses to contain either regular expressions or IP address ranges respectively. In these cases, requests contain `IPSingle` and `StringSingle` values, and clauses contain `IPRange` and `StringRange` values—both of which contain a range attribute, which is the set of values either in the IP range or that match the regular expression.

```
one sig ReceiveMessage, SendMessage, DeleteMessage,
  ChangeMessageVisibility, GetQueueAttributes extends Action {}
one sig CurrentTime, SecureTransport, SourceIP, UserAgent extends Key {}
abstract sig Boolean extends Value {}
one sig True, False extends Boolean {}
sig DateTime extends Value {}
sig IPSingle extends Value {}
sig IPRange extends Value { range : some IPSingle }
sig StringSingle extends Value {}
sig StringRange extends Value { range : some StringSingle }
```

The condition types depend on the types of values associated with the keys used in the clauses. For Boolean values there is a single condition type called `Bool`, which represents logical equivalence. For date-time values, there are condition types for comparing date-times include equality, less than and greater than. For IP addresses, there are condition types for being within or outside an IP range. Similarly, strings have condition types for matching regular expressions, as well as equality. In order to provide ordering for date-times, a standard Alloy ordering is applied to the `Value` signature.

The definitions of three of the condition types are given below.

```
one sig DateEquals extends CondType {} { match = { d1 , d2 : DateTime | d1 = d2 } }
one sig DateLessThan extends CondType {} { match = { d1 , d2 : DateTime | lt[d1,d2] } }
one sig IPAddress extends CondType {} {
  match =  { ip1 : IPSingle , ip2 : IPRange | ip1 in ip2.range}
}
```

As well as specifying the actions, condition types and values in a policy. The SQS policy documentation also adds some constraints to the policies. These include uniqueness of identifiers and limiting each policy to a single resource. These constraints are added as facts in the Alloy model. In addition, a fact has been added to ensure that the values returned by the keys function are of the correct type.

```
fact TypedKeys {
  Request.keys[CurrentTime] in DateTime
  Request.keys[SecureTransport] in Boolean
  Request.keys[SourceIP] in IPSingle
  Request.keys[UserAgent] in StringSingle
}
```

5 Examples

In this section we give two examples of use of the Alloy model. In the first, the properties of multiple policies are tested to find potential problems caused by the use of distributed access control. In the second, we look at the feasibility of using Alloy to generate candidate policies based on a set of test cases.

5.1 Two-policy test

In this example it is assumed that there are two queues: the first is used by an application to submit jobs to an EC2 instance. The results of the jobs are placed by the EC2 instance into a second queue which is then read by the application. The use of two queues in this way is one of the basic scenarios described in the SQS documentation.

There are exactly two principals used in this example: `appPrincipal` and `ec2Principal`. The resources are restricted to the two queues `requestQueue` and `reponseQueue`.

The policy for the first queue (`policy1`) contains two statements: the first allows the application to send messages providing the current date-time is less than some unspecified value; the second allows the EC2 instance to receive messages with an identical restriction on date-time values. The policy for the second queue (`policy2`) is identical to the first with the principals reversed, so the EC2 instance can send and the application can receive.

```
-- Actors
one sig appPrincipal, ec2Principal extends Principal {}
one sig requestQueue, responseQueue extends Resource {}
one sig pid1, pid2 extends PolicyId {}
one sig sid1, sid2, sid3, sid4 extends StatementId {}
-- Policy
one sig clause1 extends Clause {} { key = CurrentTime }
one sig condition1 extends Condition {} { type = DateLessThanEquals && clauses = clause1 }
one sig statement1 extends Statement {} {
  sid = sid1 && effect = Allow && principals = appPrincipal
  actions = SendMessage && resources = requestQueue && conditions = condition1
}
one sig statement2 extends Statement {} {
  sid = sid2 && effect = Allow && principals = ec2Principal
  actions = ReceiveMessage && resources = requestQueue && conditions = condition1
}
one sig policy1 extends Policy {} {
  version = v1 && pid = pid1
  statements = statement1 + statement2
}
one sig statement3 extends Statement {} {
  sid = sid3 && effect  = Allow && principals = ec2Principal
  actions = SendMessage && resources = responseQueue && conditions = condition1
}
one sig statement4 extends Statement {} {
  sid = sid4 && effect  = Allow && principals = appPrincipal
  actions = ReceiveMessage && resources = responseQueue && conditions = condition1
}
one sig policy2 extends Policy {} {
  version = v1 && pid = pid2
  statements = statement3 + statement4
}
```

To test the policies, we assume that two requests are made: the first is a request to send a message to the request queue; the second is to receive a message from the response queue. We then test a predicate that states that there exists a principal who is permitted to perform the first request but is not permitted to perform the second request. That is to say that there exists a principal who can send a message but cannot subsequently read the results.

```
one sig request1 extends Request {} { action = SendMessage && resource = requestQueue }
one sig request2 extends Request {}  { action =  ReceiveMessage && resource = responseQueue }
pred WriteButNotRead {
  request1.principal = request2.principal
  lt[request1.keys[CurrentTime],request2.keys[CurrentTime]]
  EvaluatePolicy[request1, policy1] = Allow
  EvaluatePolicy[request2, policy2] != Allow
}
```

The analyzer can find a counter-example to the predicate but requires at least 6 values, which include, `True`, `False`, an `IPSingle`, a `StringSingle` and two `DateTime` values. The `IPSingle` and `StringSingle` values are required as there must be a value for each of the four request keys. The two `DateTime` values are needed to meet the constraint that the first request happens before the second request. The unconstrained entities of the instance are represented in tree format below.

```
clause1$0
    field key
        CurrentTime$0
    field values
        DateTime$1

request1$0
    field keys
        CurrentTime$0 -> DateTime$1
        SecureTransport$0 -> False$0
        SourceIP$0 -> IPSingle$0
        UserAgent$0 -> StringSingle$0
    field principal
        appPrincipal$0

request2$0
    field keys
        CurrentTime$0 -> DateTime$0
        SecureTransport$0 -> True$0
        SourceIP$0 -> IPSingle$0
        UserAgent$0 -> StringSingle$0
    field principal
        appPrincipal$0
```

As can be seen, one failure case is that the first request is made at the same date-time as the value in the clause. As the second request must come after the first, it will fail. It should be noted that in this instance, `DateTime$1` comes before `DateTime$0` in the ordering.

5.2 Policy creation

In this example, the assumption is that a user is trying to construct a policy called `candidate` that meets three conditions: that only the application principal can read from `queue1`; that only the EC2 principal can send messages to `queue1`; and that nobody should be allowed to delete messages, change message visibility or get queue attributes from any queue.

```
assert OnlyAppCanReceive {
  all r : Request | r.action != ReceiveMessage || r.resource != queue1 ||
    (r.principal = appPrincipal <=> EvaluatePolicy[r, candidate] = Allow)
}
```

```
assert OnlyEc2CanSend {
  all r : Request | r.action != SendMessage || r.resource != queue1 ||
    (r.principal = ec2Principal  <=> EvaluatePolicy[r, candidate] = Allow)
}

assert noDeleteChangeAttr {
  no r : Request |
    r.action in DeleteMessage + ChangeMessageVisibility + GetQueueAttributes &&
    EvaluatePolicy[r, candidate] = Allow
}
```

We could use the Alloy Analyzer to find a policy that meets all of the assertions—but it is more likely to find a set of requests that avoided the constraints than a policy that met them in all cases. Instead, our approach involves creating a set of concrete examples for which the desired result was known. Two sets of requests were created: one set, which should always be successful and another set that should always fail.

```
one sig request1 extends Request {} { action =  ReceiveMessage && principal = appPrincipal }
one sig request2 extends Request {} { action = SendMessage && principal = ec2Principal  }
fun success : Request { request1 + request2 }

one sig request3 extends Request {} { action =  ReceiveMessage && principal = ec2Principal  }
one sig request4 extends Request {}  { action = SendMessage && principal = appPrincipal  }
fun failure : Request { request3 + request4 }

pred FindPolicy(generated : Policy) {
  all r : success | EvaluatePolicy[r, generated] = Allow
  all r : failure | EvaluatePolicy[r, generated] != Allow
}
```

The `FindPolicy` predicate can be used to find a candidate policy which gives the correct results for the examples given. The candidate policy can then be tested against the three original constraints to find examples of requests for which it gives incorrect results. These requests can then be added to the list of examples and a new candidate policy generated. For example, after the first iteration, the following request should have resulted in success but did not; as such, it was added to the set of successful requests.

```
one sig request5 extends Request {} {
  action = ReceiveMessage && principal = ec2Principal && resource = queue1
  keys[CurrentTime] = dateTime0 && keys[SecureTransport] = True
  keys[SourceIP] = ipSingle0 && keys[UserAgent] = stringSingle0
}
```

After six iterations, a candidate policy is found for which all three predicates hold when tested up to a size of 20.

```
one sig sid0 extends StatementId {}
one sig statement0 extends Statement {} {
  actions = ReceiveMessage
  conditions = none
  effect = Allow
  principals = appPrincipal
  resources = resource0
  sid = sid0
}
```

```
one sig sid1 extends StatementId {}
one sig statement1 extends Statement {} {
  actions = SendMessage
  conditions = none
  effect = Allow
  principals = ec2Principal
  resources = resource0
  sid = sid1
}

one sig pid0 extends PolicyId {}
one sig candidate extends Policy {} {
  pid = pid0
  version = v1
  statements = statement0 + statement1
}
```

This approach clearly has its limitations: in the general case, there is no guarantee that the number of requests required to generate a policy that meets the constraints will be of a practical size. However, if it is possible to create a policy that meets the constraints which contains a small number of statements and conditions, then the required number of tests is likely to be sufficiently small. Certainly, we envisage this test-based approach as being accessible to policy writers, and having the potential to provide some degree of formal assurance in policy construction.

6 Discussion

Cloud computing is a new computing paradigm which is gaining popularity. Computing systems built within a cloud infrastructure are constructed from multiple interacting sub-components, and access control languages can be used to restrict the interaction between these sub-components.

We have built formal models of the access policy language used within the Amazon Web Services cloud computing infrastructure. Specifically we have explored policies written for the Simple Queue Service. Using the Alloy Analyzer we have been able to explore properties of specific combinations of policies. We have also been able to use the Alloy Analyzer to assist in the construction of new policies by using sets of requests which result in known access control decisions.

As access control decisions are a security-critical function it is important that policy writers have some degree of assurance with respect to their correctness. Previous work in this area has centred around simple access control systems such as Role-Based Access Control [9, 10]. Attempts at modelling the significantly more complex XACML (see, for example, [11], [4], and [12]) have all resulted in partial models which avoid some of the more complex elements such as conditions and/or XPATH queries. In this paper we have presented a model of the whole language (which is, admittedly, less complex than XACML) that is suitable for analysis. By providing a model of the whole language it becomes possible to analyse existing real-world systems instead of placing restrictions on future systems so that analysis will be possible. This gives the model wide applicability to the rapidly increasing number of systems built using Amazon Web Services.

While the Alloy Analyzer can find instances that meet a complex set of predicates, it is not capable of simultaneously finding an instance of a policy and testing its properties against all possible requests of bounded size. Instead it will deliberately avoid requests that would break the predicates and a hence will find a policy of limited application. By separating the two steps, it is possible to build up a set of requests that will test different aspects of the predicates resulting in increasingly applicable candidate policies. By testing candidate policies against the original predicates new requests can be found which test different aspects of the predicates.

One area of further work is to combine the model of the AWS policy language with similar models for RBAC and XACML. This will give the opportunity to explore the relationships between the models, with an ultimate goal of the automatic translation of policies from one language to another. It will also be useful when modelling the types of complex heterogeneous systems that are encouraged by cloud computing. A second avenue of further work is to consider the relationship between the Alloy and Z models (with a more concrete Z model capturing the specifics of SQS policies). In particular, we will explore the mutual benefits to be afforded through a combination of model finding and theorem proving.

References

1. Weiss, A.: Computing in the Clouds. netWorker **11**(4) (2007) 16–25
2. Amazon.com: Amazon Web Services. http://aws.amazon.com/ (2009)
3. Amazon.com: Amazon Simple Queue Service Developer Guide (API Version 2009-02-01). http://docs.amazonwebservices.com/AWSSimpleQueueService/2009-02-01/SQSDeveloperGuide (2009)
4. Zhang, N., Guelev, D.P., Ryan, M.: Synthesising verified access control systems through model checking. Journal of Computer Security **16**(1) (2007) 1–61
5. Bryans, J., Fitzgerald, J.S.: Formal Engineering of XACML Access Control Policies in VDM++. In Butler, M., Hinchey, M.G., Larrondo-Petrie, M.M., eds.: ICFEM. Volume 4789 of Lecture Notes in Computer Science., Springer (2007) 37–56
6. Woodcock, J., Davies, J.: Using Z: specification, refinement, and proof. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1996)
7. Jackson, D.: Alloy: a lightweight object modelling notation. ACM Transactions on Software Engineering Methodologies **11**(2) (2002) 256–290
8. Godik, S., Moses, T.: Extensible Access-Control Markup Language (XACML) version 1.0. Technical report, OASIS (2003)
9. Zao, J., Wee, H., Chu, J., Jackson, D.: RBAC schema verification using lightweight formal model and constraint analysis. In: Proceedings of 8th ACM symposium on Access Control Models and Technologies (SACMAT). (2003)
10. Power, D.J., Slaymaker, M.A., Simpson, A.C.: On formalizing and normalizing role-based access control systems. The Computer Journal **52**(3) (2009) 305–325
11. Bryans, J.: Reasoning about XACML policies using CSP. In: Proceedings of the 2005 Workshop on Secure Web Services. (2005) 28–35
12. Hughes, G., Bultan, T.: Automated verification of access control policies using a SAT solver. International Journal on Software Tools for Technology Transfer (STTT) **10**(6) (2008) 503–520

A Z model

$[PolicyID]$
$Version ::= v1$

Policy

$version : Version$
$id : PolicyID$
$statements : \mathbb{P}_1\ Statement$

$[StatementID, Principal, Action, Resource]$
$Effect ::= Allow \mid Deny \mid SoftDeny$

Statement

$sid : StatementID$
$effect : Effect$
$principals : \mathbb{P}_1\ Principal$
$actions : \mathbb{P}_1\ Action$
$resources : \mathbb{P}_1\ Resource$
$conditions : \mathbb{P}\ Condition$

Condition

$type : CondType$
$clauses : \mathbb{P}_1\ Clause$

$[Key, Value]$

Clause

$key : Key$
$values : \mathbb{P}_1\ Value$

CondType

$match : Value \leftrightarrow Value$

Request

$principal : Principal$
$action : Action$
$resource : Resource$
$keys : Key \rightarrow Value$

$$MeetsCondition : Request \leftrightarrow Condition$$

$$\begin{aligned}
&MeetsCondition = \\
&\quad \{r : Request;\ c : Condition \mid \\
&\qquad (\forall\, cl : c.clauses \bullet \\
&\qquad\quad (\exists\, v : cl.values \bullet (r.keys(cl.key), v) \in c.type.match))\}
\end{aligned}$$

$$MatchStatement : Request \leftrightarrow Statement$$

$$\begin{aligned}
&MatchStatement = \\
&\quad \{r : Request;\ s : Statement \mid \\
&\qquad r.principal \in s.principals \land \\
&\qquad r.action \in s.actions \land \\
&\qquad r.resource \in s.resources \land \\
&\qquad (\forall\, c : s.conditions \bullet (r, c) \in MeetsCondition)\}
\end{aligned}$$

$$EvalStatement : Request \to Statement \to Effect$$

$$\begin{aligned}
&\forall\, r : Request \bullet EvalStatement(r) = \\
&\quad (\lambda\, s : Statement \bullet SoftDeny) \\
&\quad \oplus \\
&\quad (\lambda\, s : Statement \mid (r, s) \in MatchStatement \bullet s.effect)
\end{aligned}$$

$$EvalPolicy : Request \to Policy \to Effect$$

$$\begin{aligned}
&\forall\, r : Request \bullet EvalPolicy(r) = \\
&\quad (\lambda\, p : Policy \bullet SoftDeny) \\
&\quad \oplus \\
&\quad (\lambda\, p : Policy \mid \\
&\qquad (\exists\, s : p.statements \bullet EvalStatement(r)(s) = Allow) \bullet Allow) \\
&\quad \oplus \\
&\quad (\lambda\, p : Policy \mid \\
&\qquad (\exists\, s : p.statements \bullet EvalStatement(r)(s) = Deny) \bullet Deny)
\end{aligned}$$

$$EvalPolicies : Request \to (\mathbb{P}\, Policy) \to \{Allow, Deny\}$$

$$\begin{aligned}
&\forall\, r : Request \bullet EvalPolicies(r) = \\
&\quad (\lambda\, pols : \mathbb{P}\, Policy \bullet Deny) \\
&\quad \oplus \\
&\quad (\lambda\, pols : \mathbb{P}\, Policy \mid \\
&\qquad (\exists\, p : pols \bullet EvalPolicy(r)(p) = Allow) \bullet Allow) \\
&\quad \oplus \\
&\quad (\lambda\, pols : \mathbb{P}\, Policy \mid \\
&\qquad (\exists\, p : pols \bullet EvalPolicy(r)(p) = Deny) \bullet Deny)
\end{aligned}$$

B Basic alloy model

```
module AWS/policy

-- Domain specific abstract signatures
abstract sig Value, Key, Action {}
abstract sig CondType {
  match : Value -> Value
}

-- General signatures
sig Clause {
  key : Key ,
  values : some Value
}

sig Condition {
  type : CondType ,
  clauses : some Clause
}

sig StatementId, Principal, Resource {}
sig Statement {
  sid : StatementId ,
  effect : Effect ,
  principals : some Principal ,
  actions : some Action ,
  resources : some Resource ,
  conditions : set Condition
}

abstract sig Effect {}
one sig Allow, Deny, SoftDeny extends Effect {}

sig PolicyId {}
abstract sig Version {}
one sig v1 extends Version {}
sig Policy {
  version : Version ,
  pid : PolicyId ,
  statements : some Statement
}

sig Request {
  principal : Principal ,
  action : Action ,
  resource : Resource ,
  keys : Key -> one Value
}

-- Evaluation logic
pred MeetsCondition ( r : Request , c : Condition ) {
  all cl : one c.clauses | some v : one cl.values | (r.keys[cl.key] -> v) in c.type.match
}

pred MatchStatement ( r : Request , s : Statement ) {
  r.principal in s.principals
  r.action in s.actions
  r.resource in s.resources
  all c : one s.conditions | MeetsCondition[r,c]
}

fun EvalStatement ( r : Request , s : Statement ) : Effect {
  MatchStatement[r,s] => s.effect else SoftDeny
}
```

```
fun EvalPolicy ( r : Request , p : Policy ) : Effect {
  (some s : one p.statements | EvalStatement[r,s] = Deny) => Deny else
  ((some s : one p.statements | EvalStatement[r,s] = Allow) => Allow else SoftDeny)
}

fun EvalPolicies ( r : Request , ps : set Policy ) : Effect {
  (some p : one ps | EvalPolicy[r,p] = Deny) => Deny else
  ((some p : one ps | EvalPolicy[r,p] = Allow) => Allow else Deny)
}
```

C SQS alloy model

```
module AWS/sqs
open AWS/policy
open util/ordering [Value]

-- Request Keys
one sig CurrentTime , SecureTransport , SourceIP , UserAgent extends Key {}

-- Actions
one sig ReceiveMessage , SendMessage , DeleteMessage ,
  ChangeMessageVisibility , GetQueueAttributes extends Action {}

-- Facts
fact UniquePolicyIds{
  all p1,p2 : Policy | p1 = p2 iff p1.pid = p2.pid
}
fact UniqueStatementIds{
  all s1,s2 : Statement | s1 = s2 iff s1.sid = s2.sid
}
fact OneResourcePolicies {
  all p : Policy | lone p.statements.resources
}

--   Boolean values
abstract sig Boolean extends Value {}
one sig True , False extends Boolean {}
--   DateTime values
sig DateTime extends Value {}
-- IP values
sig IPSingle extends Value {}
sig IPRange extends Value {
  range : some IPSingle
}
--   String values
sig StringSingle extends Value {}
sig StringRange extends Value {
  range : some StringSingle
}

-- Constrain keys function of requests to return values of correct types
fact TypedKeys {
  Request.keys[CurrentTime] in DateTime
  Request.keys[SecureTransport] in Boolean
  Request.keys[SourceIP] in IPSingle
  Request.keys[UserAgent] in StringSingle
}

-- Conditions
one sig Bool extends CondType {} {
  match = { b1 , b2 : Boolean | b1 = b2 }
}
one sig DateEquals extends CondType {} {
  match = { d1 , d2 : DateTime | d1 = d2 }
}
one sig DateNotEquals extends CondType {} {
  match = { d1 , d2 : DateTime | d1 != d2 }
}
one sig DateLessThan extends CondType {} {
  match = { d1 , d2 : DateTime | lt[d1,d2] }
}
one sig DateLessThanEquals extends CondType {} {
  match = { d1 , d2 : DateTime | lte[d1,d2] }
}
one sig DateGreaterThan extends CondType {} {
  match = { d1 , d2 : DateTime | gt[d1,d2] }
}
```

```
one sig DateGreaterThanEquals extends CondType {} {
  match = { d1 , d2 : DateTime | gte[d1,d2] }
}
one sig IPAddress extends CondType {} {
  match =  { ip1 : IPSingle , ip2 : IPRange | ip1 in ip2.range}
}
one sig NotIPAddress extends CondType {} {
  match =  { ip1 : IPSingle , ip2 : IPRange | ip1 not in ip2.range}
}
one sig StringEquals extends CondType {} {
  match = { s1 , s2 : StringSingle | s1 = s2 }
}
one sig StringNotEquals extends CondType {} {
  match = { s1 , s2 : StringSingle | s1 != s2 }
}
one sig StringLike extends CondType {} {
  match = { s1 : StringSingle , s2 : StringRange | s1 in s2.range}
}
one sig StringNotLike extends CondType {} {
  match = { s1 : StringSingle , s2 : StringRange | s1 not in s2.range}
}
```