Conformance checking of dynamic access control policies

David Power, Mark Slaymaker and Andrew Simpson

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

Abstract. The capture, deployment and enforcement of appropriate access control policies are crucial aspects of many modern software-based systems. Previously, there has been a significant amount of research undertaken with respect to the formal modelling and analysis of access control policies; however, only a limited proportion of this work has been concerned with *dynamic* policies. In this paper we explore techniques for the modelling, analysis and subsequent deployment of such policies—which may rely on external data. We use the Alloy modelling language to describe constraints on policies and external data; utilising these constraints, we test static instances constructed from the current state of the external data. We present Gauge, a constraint checker for static instances that has been developed to be complementary to Alloy, and show how it is possible to test systems of much greater complexity via Gauge than can typically be handled by a model finder.

1 Introduction

Large-scale data-oriented systems dominate much of our lives: as employees, as consumers, as patients, as travellers, as web surfers, and as citizens. The nature of much of this data, coupled with an increased awareness of relevant security and privacy issues, means that it is essential that effective tools, technologies and processes are in place to ensure that any and all access is appropriate. Our concern in this paper is the construction of access control policies that rely on context to inform decisions. (Arguments as to the potential benefits of *context-sensitive access control* have been made by, for example, [1], [2], and [3].) Specifically, our concern is what might be termed *evolving access control*—whereby access control decisions are made on the basis of state.

We utilise formal models for the construction and analysis of such dynamic policies. In this respect, our work has much in common with that of [4], which defines a framework to capture the behaviour of access control policies in dynamic environments. (In common with our approach, the authors also separate the policy from the environment.) Importantly, our work is driven by practical concerns. The policies that are constructed and analysed are subsequently deployed to instances of the *sif* (service-oriented interoperability framework) middleware framework [5, 6] to support the secure sharing and aggregation of data.

The framework supports relatively straightforward policies that conform to the role-based access control (RBAC) model [7]; it also supports more complex policies in the expressive XACML (eXtensible Access Control Markup Language) policy language.[1]

Of course, the use of access control policies can bring many benefits when managing complex systems: by centralising all authorisation decisions, consistency of access can be maintained, and updating a single access control policy is much simpler than modifying multiple components. Nevertheless, creating and updating access control policies is still a potentially time-consuming task. Going further, policy languages such as XACML support access to external data—which may be updated independently of the policy. While this simplifies the task of maintaining policies, it greatly complicates their analysis and also necessitates controls on the modification of external data.

As demonstrated by many authors, formal methods have a role to play in this area, with examples including the work of [8] and [9]—both of which are concerned with the modelling and analysis of XACML. Even when the requirements for an access control policy are well understood, it is still possible for mistakes to be made: the flexibility of policy languages increases the potential for mistakes due, in part, to their expressiveness.

We utilise the Alloy Modelling Language [10] in this paper to build models of policies and external data. Using the Alloy Analyzer we are able to test properties of those models. By constructing instances of policies and external data, we are able to evaluate the constraints described in the Alloy model.[2] However, the Alloy Analyzer is only capable of analysing models of bounded size; this and a lack of support for the large integers needed to model times, dates and monetary values has led some researcher to avoid using the Alloy Modelling Language [16]. To address these problems, we have developed a tool for checking constraints on large policies which also has the potential to support large integers and other data types.

While, in general, it is not possible to say if a policy is 'correct' (due to the 'safety problem' of [17]), it is possible to test for certain healthiness conditions, such as separation of duty constraints in role-based policies. Of course, there are many other possible constraints which may be appropriate in role-based policies, such as the absence (or presence) of a user with all permissions, or all users having at least one role.

To this end, we concern ourselves with RBAC models and policies as a means of illustrating the contribution. Specifically, we build on the RBAC model of [18], which has been utilised in the policy editing tool described in [19]. It should be noted that the modelling and analysis of RBAC constraints has a rich history, with the work of [20] and [21] being of particular note.

---

[1] See `http://www.oasis-open.org/committees/xacml/`.

[2] Other work that has built policy analysis tools on Alloy include the contributions of [11], [12] and [13]. Also relevant in terms of related work is the DynAlloy tool [14, 15], which extends Alloy to handle dynamic specifications.

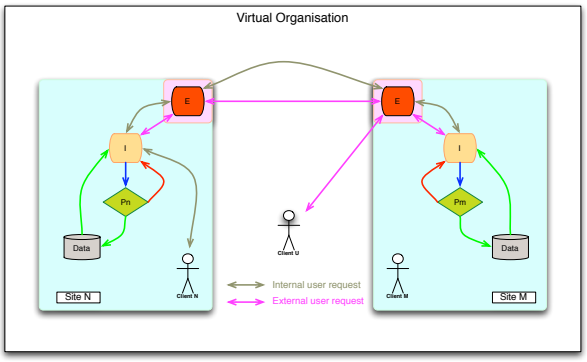

Fig. 1. The sif view of a distributed system

The structure of the remainder of this paper is as follows. In Section 2 we describe the motivation for, and context of, our work: the capture, analysis and enforcement of dynamic access control policies that make reference to external state. Then, in Section 3, we describe the modelling and analysis of constraints via Alloy. In Section 4 we introduce Gauge—our tool for the evaluation of Alloy predicates and expressions. Finally, in Section 5, we summarise the contribution of this paper and outline potential areas of future work.

2 Context

In this section we present the background to our work. We start by introducing the sif framework, before giving consideration to what we term *evolving access control*. We then briefly introduce our RBAC policy editing tool.

2.1 sif

sif (service-oriented interoperability framework) is concerned with supporting secure data sharing and aggregation in a fashion that doesn't require organisations to throw away existing data models or systems, change practices, or invest in new technology. The philosophy behind sif was originally described in [22]. There, a virtual organisation—spread across two or more geographically or physically distinct units—was characterised as per Figure 1. Deployments communicate via their external interfaces (represented by E), with data being accessed via an internal interface, I. The permitted access to the data is regulated by policies (P): each organisation has control over its data, which means that the responsibility for defining policies resides a local level.

sif offers support for three types of 'plug-in'—data plug-ins, file plug-ins and algorithm plug-ins—and it is these plug-ins that facilitate interoperability. By using a standard plug-in interface, it becomes possible to add heterogeneous

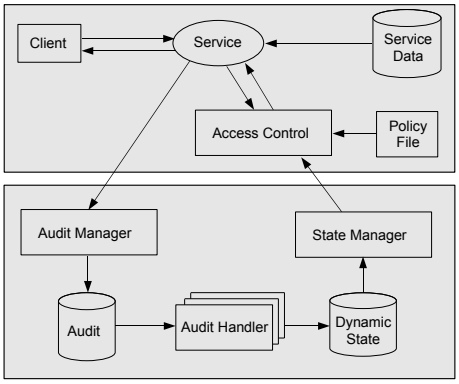

Fig. 2. Evolving access control architecture

resources into a virtual organisation. If, in a distributed, heterogeneous context, a user runs a query across several data nodes, then the middleware will distribute that query to the nodes and aggregate the results. The middleware exposes as much to the user as the developer considers useful for the application in question: it may be appropriate to expose the whole underlying data structure, allowing users to construct SQL queries; alternatively, a simple interface supporting pre-formulated queries might be appropriate.

2.2 Evolving access control

The middleware framework of the previous section has the potential to support what might be termed *evolving access control*: what may be accessed by users and applications may change dynamically, depending on context. Examples of such policies might include "if there has been no contact from Officer X for over 30 minutes then access should be denied from her device," "Professor Y can access up to 10 of these images," and "Dr Z can access data provided that the network capacity is sufficient." *Meta-policies* prescribe the relationship between policies: after Officer X has been out of contact for over 30 minutes, any access that was previously possible is now denied; once Professor Y has accessed 10 images, she can access no more of them; when the network capacity increases, Dr Z can access data to his heart's content.[3] Thus, these meta-policies are necessarily written at a higher level of abstraction than policies—and, as such, are intended to be closer to the level at which requirements might be captured or guidelines might be stated.

[3] Note that our notion of meta-policy—describing the relationship between policies—differs from that of [23]—where the concern is 'policies about policies'.

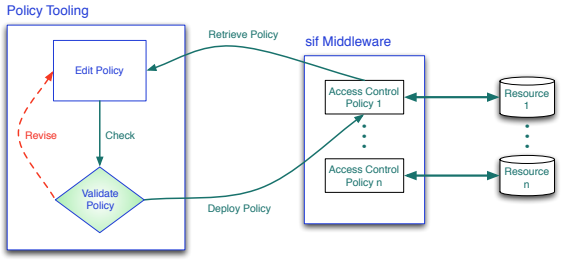

Fig. 3. Policy editing and validation workflow

Of course, providing a system with the functionality to adapt access control policies automatically means that the need for assurance that the *correct* policy is in place necessarily increases: ensuring that certain fundamental properties hold in every potential state, for example, is essential—we would not want our protection mechanism to evolve into a state that provided little protection, for example. Hence the driver for a *model-driven* approach to evolving access control: raising the level of abstraction for data owners and policy writers, with a view to giving a degree of assurance that data sharing is appropriate.

In this paper, our concern is the modelling and analysis of external data referenced by access control policies. Ideally, it would be possible not only to analyse the current state of external data but to model the modifications of that data. This is possible with our evolving access control deployments, where the combination of policy and external state evolve in accordance with a meta-policy.

The architecture of the existing evolving access control system is presented in Figure 2. Here, an audit of client activity and other events of significance is handled by an audit manager. Handlers monitor the audit and update the dynamic state in accordance with the rules of the meta-policy. When the server makes an access control request, external data can be accessed via a state manager which reads the dynamic state.

2.3 RBAC policy editing tool

The sif middleware can support a number of different types of access control policies, with the associated RBAC editing tool of [19] allowing the creation and modification of RBAC policies. An illustration of how formal modelling and analysis is incorporated into the overall RBAC policy workflow is given in Figure 3.

Once modified, the policy is converted into an Alloy instance and then validated against an Alloy model using the Alloy Evaluator. The tool tests each policy against 12 different constraints; if a constraint does not hold, the user is informed of the reason why so that the policy can be revised. The process of creating instances is discussed in more detail in Section 4, in which we also describe an alternative method of evaluating constraints.

3 Constraints and requirements

The work described in the following is motivated by the desire to be able to ensure that policy level constraints hold in the presence of dynamically changing data—assuming that we are aware of potential changes to the external data and, as such, can perform constraint checking before the changes are made. The simplest solution to this problem is to treat the entire policy as external data and to check all of the constraints whenever there is a potential change. However, a more efficient approach is to check just the constraints that are dependent on the external data; this approach assumes that any constraints that are independent of the external data have been checked at the time of policy construction.

We start by describing the model for RBAC policies that we leverage to illustrate our contribution.

3.1 RBAC

The underlying principle upon which role-based access control is based is the association of permissions with the roles that users may hold within an organisation. There are four standard components in the ANSI standard for role-based access control systems [7].

- *Core RBAC* is mandatory in any RBAC system, and associates permissions with roles and roles with users.
- Any combination of the following can be utilised in a particular system.
  1. *Role hierarchies* define what amounts to an inheritance relation between roles. As an example, role r_1 inherits from role r_2 if all privileges associated with r_2 are also associated with r_1.
  2. A *static separation of duty* (SSD) constraint is characterised by a role set, rs, such that $\# rs \geq 2$, and a natural number, n, such that $2 \leq n \leq \# rs$, and ensures that no user can be authorised for n or more roles in rs.
  3. A *dynamic separation of duty* (DSD) constraint is concerned with sessions: a DSD constraint ensures that no user can be associated with n or more roles in rs in a particular session.

3.2 An Alloy representation of RBAC

We present a model for core RBAC with hierarchy and static separation of duty constraints. The model is based on that of [18], which presents a formal description of RBAC using the formal description language, Z [24, 25].

First, we introduce `User`, `Role`, `Action` and `Resource`, with `Action` and `Resource` being used to define the contents of a `Permission`.

```
sig User, Role, Action, Resource {}

sig Permission {
  action : Action,
  resource : Resource
}
```

The `MER` signature represents a mutually exclusive roles constraint which restricts the combinations of roles a user can be associated with. The signature consists of an integer, `limit`, and a set of roles, `roles`. The `MER` signature also has a constraint, which states that the value of `limit` ranges between 2 and the cardinality of the `roles` set.

```
sig MER {
  limit : Int,
  roles : set Role
} {
  2 <= limit
  limit <= (# roles)
}
```

The fact `uniquePermission` ensures that each `Permission` is unique, i.e. no two different (signified by `disj`) elements of `Permission` have the same action–resource pair. This simplifies the subsequent definitions in the RBAC model.

```
fact uniquePermission {
  all disj pb1, pb2 : Permission |
    pb1.action != pb2.action || pb1.resource != pb2.resource
}
```

The `Hierarchy` signature represents an RBAC system with a role hierarchy and static separation of duty constraints. It contains the sets `USERS`, `ROLES` and `PRMS`, which represent the particular users, roles and permissions to which the policy relates. It also contains the relations `UA`, `PA` and `RH`, which represent the user-role, role-permission and role hierarchy mappings. The set `SC` is a set of static separation of duty constraints, the roles of which must be a subset of `ROLES`. The relation `RH` must be acyclic, which is ensured by `no (^RH & iden)`. The composition `UA.*RH.PA` creates a user–permission relation which relates users and their reachable permissions taking into account the role hierarchy.

```
sig Hierarchy {
  USERS :  set User,
  ROLES : set Role,
  PRMS : set Permission,
  UA : USERS -> ROLES,
  RH : ROLES -> ROLES,
  PA : ROLES -> PRMS,
  SC : set MER
} {
  no (^RH & iden)
  all s : SC | s.roles in ROLES
}
```

We now describe a number of constraints that can be used to validate policies. The applicability of each constraint will depend, of course, upon the context of

the deployed policy. In total, there are 12 constraints that are checked by our RBAC policy construction tool, three of which are presented below.

The first example is a constraint on any individual user having all permissions, represented as a fact called `NobodyCanDoEverything` affecting all elements of `Hierarchy`. This fact could have been included in the signature of `Hierarchy`, but is written as a separate fact to promote modularity and (consequently) to allow it to be tested independently.

```
fact NobodyCanDoEverything {
  all h : Hierarchy, u : h.USERS |
    u.(h.UA).*(h.RH).(h.PA) != h.PRMS
}
```

Similarly, the enforcement of static separation of duty constraints is written as a separate fact called `NobodyBreachesSC`. If the constraint does not hold, the tool evaluates the function `fun_NobodyBreachesSC` which returns a set containing (`Hierarchy`, `MER`, `User`) triples indicating, for each hierarchy, the particular static separation of duty of constraint which has been breached and the user that breaches it. Similar functions exist for the other constraints.

```
fact NobodyBreachesSC {
  all h : Hierarchy, s : h.SC, u : h.USERS |
      #(s.roles & u.(h.UA).*(h.RH)) < s.limit
}

fun fun_NobodyBreachesSC() : Hierarchy -> MER -> User {
  { h : Hierarchy, s : h.SC, u : h.USERS |
      #(s.roles & u.(h.UA).*(h.RH)) >= s.limit }
}
```

There are also constraints relating to redundancy in the model. One such example is `NoRedundantPermissions`, which prevents a role from being assigned a permission that it already holds due to inheritance.

```
fact NoRedundantPermissions {
  all h : Hierarchy, r : h.ROLES |
    no (r.(h.PA) & r.^(h.RH).(h.PA))
}
```

3.3 Adding sessions

We now consider how sessions can be added to the RBAC model so as to allow us to divide a policy into static and dynamic parts. We assume that the dynamic parts of the policy are stored as external data.

The signature `Session` extends `Hierarchy`. The relation `AR` contains the currently active roles for each user; this represents the dynamic part of the policy. The set `DC` is a set of dynamic separation of duty constraints, which restrict the

active roles of a user (it is assumed that the set DC does not change dynamically). As was the case for static separation of duty constraints, it is assumed that all dynamic separation of duty constraints refer only to roles from the set ROLES. The composition AR.PA is now used to relate users to their current permissions.

```
sig Session extends Hierarchy {
   AR : USERS -> ROLES,
   DC : set MER
} {
   all d : DC | d.roles in ROLES
}
```

The value of the relation AR needs to meet two criteria: each user–role pair has to represent a role that the user has access to, and the dynamic separation of duty constraints must be met. These criteria are captured in the fact DynamicFact.

```
fact DynamicFact {
   all s : Session |
     s.AR in (s.UA).*(s.RH) &&
     all d : s.DC , u : s.USERS |
       #(d.roles & u.(s.AR)) < d.limit
}
```

As none of the constraints on Hierarchy make reference to AR they will not need to be checked when AR changes, which reduces the amount of dynamic constraint checking required. It is possible to add extra constraints that do depend on the AR relation. For example, it is possible to modify the fact NobodyCanDoEverything to only depend on the currently activated roles.

```
fact NobodyCanCurrentlyDoEverything {
  all s : Session, u : s.USERS |
    u.(s.AR).(s.PA) != s.PRMS
}
```

4 Gauge

In this section we discuss Gauge, a means of evaluating Alloy predicates and expressions that has been developed as a companion tool to the Alloy Analyzer. Unlike the Alloy Analyzer, Gauge is not a model finder and can only work with known instances. Specifically, Gauge is designed for instances built from real world data which are too large for a model finder to handle; it is also capable of handling large integers and other data types which are commonly found in practice.

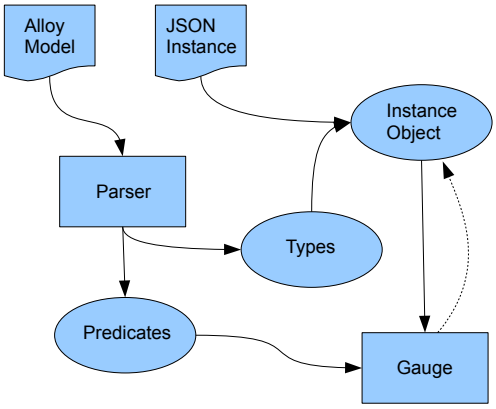

Fig. 4. Parsing and evaluation process

4.1 Overview

The first stage of using Gauge involves creating an Alloy model and using the Alloy Analyzer to check the suitability of the model. Once this has been done, it is then possible to construct a JavaScript Object Notation (JSON) instance of the data using the signatures defined in the model.

Figure 4 shows how the Alloy model and JSON instance are processed. The Alloy model is parsed using a modified version of the parser used by the Alloy Analyzer; this creates both type information for the signatures and predicate information for the facts and other constraints. The type information is used to turn the JSON instance into an object representation (considered further in the next section). Once the instance object has been created, the predicates can then be evaluated. In certain circumstances, Gauge can extend the instance by adding new atoms; this is represented by the dotted arrow.

4.2 Instances

An instance consists of three types of data: atoms, signature relations and field relations. Atoms are the basic building blocks of an instance and each atom has a signature type. The signature relations are sets of atoms of a certain signature type. Where one signature extends another, atoms of the subtype will appear in both signature relations. Field relations are sets of tuples, the first element of which is the atom to which the field relates.

In a JSON instance, each atom is introduced as a separate object. The `id` field is used as a unique identifier for the atom and the `type` field represents

the signature type of the atom. If the atom has any field values, these are listed in `fields` where each field name is associated with an array of arrays of atom identifiers. The inner arrays are necessary as field values could be of any arity. Shown below is an example of JSON instance containing a `MER` atom and its associated roles.

```
{ id : role1, type : Role },
{ id : role2, type : Role },
{ id : mer1, type : MER,
  fields : {
    limit : [[2]],
    roles : [[role1],[role2]]}
}
```

When loaded, the following atoms, signature relations and field relations are created, including the `univ` signature relation which all signatures extend. Integers are identified using a decimal string representation.

$$\begin{aligned} &\mathit{Atoms} = \{\mathit{role}1, \mathit{role}2, \mathit{mer}1, 2\} \\ &\mathit{Signatures} = \{\mathit{Role} \rightarrow \{\mathit{role}1, \mathit{role}2\}, \mathit{MER} \rightarrow \{\mathit{mer}1\}, \\ &\qquad\qquad \mathit{Int} \rightarrow \{2\}, \mathit{univ} \rightarrow \{\mathit{role}1, \mathit{role}2, \mathit{mer}1, 2\}\} \\ &\mathit{Fields} = \{\mathit{limit} \rightarrow \{(\mathit{mer}1, 2)\}, \mathit{roles} \rightarrow \{(\mathit{mer}1, \mathit{role}1), (\mathit{mer}1, \mathit{role}2)\}\} \end{aligned}$$

4.3 Evaluation

Each predicate that is to be evaluated is constructed from a number of expressions. When evaluated, an expression can have either a Boolean value, a relational value, or a primitive integer value. For relational operators, such as composition (`.`) or union (`+`), Gauge first evaluates the two sub-expressions and then combines the resulting relations using the operator specified. When reference is made to a signature or field, the associated relation is retrieved from the instance.

For Boolean operators such as (`&&`) or (`||`), a 'short-circuit' approach is used whereby sub-expressions are evaluated from left to right as required. Similarly, when evaluating quantifiers such as `some` or `all` evaluation stops as soon as a definitive result is found.

Some expressions introduce variables, the simplest of which is `let`. To store the current values of variables, a mapping is maintained between variables and the relations they represent. For a `let` expression, the value of the variable is fixed within each evaluation. After evaluating the body of the `let` expression, the variable is removed from the mapping before the result is returned. For quantifiers, the value of each variable is drawn from a set, the body of the quantifier is evaluated separately for each combination of variable values. Set comprehensions work similarly to quantifiers with successful combinations of variable values being turned into tuples and added to the resulting relation.

Calls to predicates and functions are handled dynamically during evaluation. The arguments are first calculated and added to the variable mapping. The

body of the called predicate or function is then evaluated. As a natural consequence of this evaluation method, Gauge is capable of evaluating certain types of recursively defined functions and predicates.

4.4 Types

While it is possible to model specific aspects of data types when constructing an Alloy model, a certain amount of abstraction is needed if one wishes to use the model-finding capabilities of the Alloy Analyzer. An example of this is the representation of integers, where the bit width is restricted. When testing for counterexamples, the restriction of the bit width is not normally a problem; however, real world data is, of course, likely to exceed the bit width used for modelling.

There are many other data types that would be of relevance in an access control system, including times, dates, strings and X.509 certificates. Each of these would be impossible to model completely in Alloy but are simple to handle in a general purpose programming language.

Such types are handled in a straightforward manner in Gauge by casting between Alloy atoms and native representations as necessary. As a simple example, part of the Gauge time module is presented below. Here, `currentTime` refers to the current time, and the predicate `Time_lte` is used for comparisons.

```
sig Time {}
one sig currentTime extends Time {}

pred Time_lte(t1, t2 : Time) {
  lte[t1,t2]
}
```

As a simple example, the predicate `NineToFive` can be used to check if the time of evaluation is in 'normal office hours': between 9am and 5pm.

```
pred NineToFive() {
  Time_lte[T_9_0_0,currentTime]
  Time_lte[currentTime,T_17_0_0]
}
```

When evaluating `NineToFive`, Gauge will recognise the predicate `Time_lte` as a predicate on time and, instead of expanding its definition, will perform the comparison using native Java objects. For atoms such as `T_9_0_0`, Gauge will create a time object with the three numbers representing hours, minutes and seconds. The atom `currentTime` is also recognised as a special case and a new atom is created which represents the current time.

It is possible to use the same methods to allow time arithmetic, such as adding an hour or calculating the difference between two times. These will potentially add new atoms to the instance. Without the ability to add new atoms, all intermediate results would need to form part of the initial instance.

Users	Atoms	Static time	Dynamic time
256	530	41ms	8ms
512	995	88ms	14ms
1024	1925	189ms	24ms
2048	3785	544ms	110ms
4096	7505	1.78s	169ms
8192	14945	6.71s	591ms
16384	29825	27.5s	2.29s
32768	59585	122s	8.38s
65536	119105	553s	35.6s

Fig. 5. Scalability results

4.5 Scalability

While the model finder used by Alloy is capable of dealing with the case when the relations are all fixed, it still is restricted by internal data structures which put a limit on the total number of atoms of $2^{31/n}$ (where n is the largest arity of any relation in the model). In our RBAC model, the `UA`, `PA` and `RH` field relations are all of arity 3, which imposes a limit of approximately 1,000 total atoms. Gauge, on the other hand, does not have any restrictions on the size of the instance other than the memory needed to store it.

To test scalability, instances of policies were created, and the time taken to test the 12 static and 2 dynamic constraints were recorded. A simple role hierarchy was created, with all apart from one role being connected in a binary tree. One user was allocated the 'separate role', with all others being randomly allocated between one and three roles from the tree. For each role, there were exactly four permissions, each having a unique action and resource. There was a single separation of duty constraint for every 16 roles—with each such constraint involving two roles, one of which was the 'separate role'. By allocating the roles, permissions and constraints in this fashion, it was possible to ensure that all but one of the constraints held, maximising the amount of work required. The constraint that did not hold pertained to a role being senior to multiple roles.

There were 16 times as many users as roles, and 8 times as many active roles. The number of users is listed in Figure 5, together with the total number of atoms. The total number of atoms includes 64 integers and the atom representing the policy, but otherwise is proportional to the number of users.

4.6 Optimisation

To achieve the times listed in Figure 5, the constraints were modified so that they could be evaluated more efficiently. As discussed previously, Gauge uses a simple evaluation strategy for quantifiers where the body is evaluated separately for each combination of variable values. This can lead to an expression being evaluated multiple times for the same values. By using `let` statements, it is

possible to store the results of expressions so they can be reused. Shown below is `DynamicFact` rewritten using `let` statements.

```
fact DynamicFact {
  all s : Session |
    let sess = s.AR |
      sess in (s.UA).*(s.RH) &&
      all d : s.DC |
        let lim = d.limit |
          let rol = d.roles |
            all u : s.USERS | #(rol & u.sess) < lim
}
```

In this case the difference in performance is significant: with 8192 users, the evaluation took 454 seconds without the `let` statements and 0.795 seconds with the `let` statements. Other optimisations are less obvious; for example, `uniquePermission` can be rewritten to remove the quantifiers completely.

```
fact uniquePermission {
  (action.~action & resource.~resource) in iden
}
```

Again, the difference in performance is significant: with 2048 permissions, the original `uniquePermission` took 14 seconds to evaluate compared with 0.022 seconds for the alternative version.

Another potential area of performance gain when evaluating the dynamic constraints comes from using an incremental approach to evaluation. In the current example, the deactivation of a role can never result in the breaching of a constraint and the activation of a role can only result in breaches related to the user doing the activating and the role(s) being activated. A combination of storing the value of expressions related to the static part of the policy (such as `(s.UA).*(s.RH)`) and only evaluating quantifiers for the parts of the dynamic policy that have changed (removing `all u : s.USERS`) would have a dramatic effect on performance.

5 Conclusions and further work

In this paper we have discussed methods for the modelling and analysis of access control policies which reference external data. The referencing of external data is of particular relevance when dealing with dynamic access control policies which are constantly modified in response to user activity or system events. By building models of access control policies in the Alloy modelling language, we are able to describe policy constraints and test existing policies. By directly creating Alloy instances, it becomes possible to test more complex policies than might be handled by the Alloy model finder. We have described a prototype evaluator

called Gauge which is capable of handling large instances and also has limited support for real world data types.

By decomposing a model of an access control policy into static and dynamic parts, we have shown how it is possible to test just a small set of constraints when the dynamic parts of a policy change. It is possible to test the suitability of a set of constraints by using the model-finding capabilities of the Alloy Analyzer; once a set of constraints has been found to be suitable, a simple evaluation of a new instance is sufficient.

The long-term goal is to be able to analyse changes in dynamic state in real time. While this is currently feasible for policies which have hundreds or even thousands of users, the evaluation time for larger policies starts to become prohibitive. By providing a facility for the persistent storage of instances it will be possible to cache results and to only evaluate constraints related to changes in the dynamic state. With such a system in place, it should be possible to handle significantly larger policies.

For changes to the static parts of policies, there is the potential to increase the speed of evaluation by decomposing the problem and utilising parallel evaluation; with the advent of cloud computing infrastructures, such approaches are now more feasible than ever before, and we intend exploring the potential for benefitting from such developments in the near future. Finally, while the development of Gauge has been driven by the needs of a particular domain, we consider it to be a general purpose evaluator; as such, we will be giving consideration to further application areas in the coming months.

References

1. Kumar, A., Karnik, N., Chafle, G.: Context sensitivity in role-based access control. ACM SIGOPS Operating Systems Review **36**(3) (2002) 53–66
2. Bhatti, R., Bertino, E., Ghafoor, A.: A trust-based context-aware access control model for web-services. Distributed and Parallel Databases **18**(1) (2005) 83–105
3. Hulsebosch, R.J., Salden, A.H., Bargh, M.S., Ebben, P.W.G., Reitsma, J.: Context sensitive access control. In: Proceedings of the 10th ACM Symposium on Access Control Models and Technologies (SACMAT 2005). (2005) 111–119
4. Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Specifying and reasoning about dynamic access-control policies. In: Proceedings of the 3rd International Joint Conference on Automated Reasoning (IJCAR 2006), Springer-Verlag Lecture Notes in Computer Science volume 4130 (2006) 632–646
5. Simpson, A.C., Power, D.J., Russell, D., Slaymaker, M.A., Kouadri-Mostefaoui, G., Ma, X., Wilson, G.: A healthcare-driven framework for facilitating the secure sharing of data across organisational boundaries. Studies in Health Technology and Informatics **138** (2008) 3–12
6. Slaymaker, M.A., Power, D.J., Russell, D., Simpson, A.C.: On the facilitation of fine-grained access to distributed healthcare data. In: Proceedings of the 5th VLDB Workshop on Secure Data Management (SDM 2008), Springer-Verlag Lecture Notes in Computer Science volume 5159 (2008) 169–184
7. Ferraiolo, D.F., Sandhu, R.S., Gavrilla, S., Kuhn, D.R., Chandramouli, R.: Proposed NIST standard for role-based access control. ACM Transactions on Information and Systems Security **4**(3) (2001) 224–274

8. Zhang, N., Ryan, M., Guelev, D.P.: Synthesising verified access control systems in XACML. In: Proceedings of the 2nd ACM Workshop on Formal Methods in Security Engineering (FMSE 2004). (2004) 56–65
9. Bryans, J.W., Fitzgerald, J.S.: Formal engineering of XACML access control policies in VDM++. In: Proceedings of the 9th International Conference on Formal Engineering Methods (ICFEM 2007), Springer-Verlag Lecture Notes in Computer Science volume 4789 (2007) 37–56
10. Jackson, D.: Software Abstractions: Logic, Language, and Analysis. MIT Press (2006)
11. Schaad, A., Moffett, J.D.: A lightweight approach to specification and analysis of role-based access control extensions. In: Proceedings of the 7th ACM Symposium on Access Control Models and Technologies (SACMAT 2002). (2002) 13–22
12. Hughes, G., Bultan, T.: Automated verification of access control policies. Technical Report 2004-22, University of California, Santa Barbara (2004)
13. Fisler, K., Krishnamurthi, S., Meyerovich, L., Tshantz, M.C.: Verification and change-impact analysis of access-control policies. In: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005). (2005) 196–205
14. Frias, M.F., Galeotti, J.P., Pombo, C.G.L., Aguirre, N.M.: DynAlloy: upgrading Alloy with actions. In: Proceedings of the 27th International Conference on Software Engineering (ICSE 2005). (2005) 442–451
15. Frias, M.F., Pombo, C.G.L., Galeotti, J.P., Aguirre, N.M.: Efficient analysis of DynAlloy specifications. ACM Transactions on Software Engineering and Methodology (TOSEM) **17**(1) (2007) Article number 4
16. Shaikh, R.A., Adi, K., Logrippo, L., Mankovski, S.: Inconsistency detection method for access control policies. In: Proceedings of 6th International Conference on Information Assurance and Security (IAS 2010). (2010) 204–209
17. Harrison, M.A., Ruzzo, W.L., Ullman, J.D.: Protection in operating systems. Communications of the ACM **19**(8) (1976) 461–471
18. 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
19. Power, D.J., Slaymaker, M.A., Simpson, A.C.: Automatic conformance checking of role-based access control policies via Alloy. In: Proceedings of the 2011 International Symposium on Engineering Secure Software and Systems (ESSoS 2011), Springer-Verlag Lecture Notes in Computer Science volume 6542 (2011) 15–28
20. Ahn, G.J., Sandhu, R.S.: Role-based authorization constraint specification. ACM Transactions on Information and Systems Security **3**(4) (2000) 207–226
21. Crampton, J.: Specifying and enforcing constraints in role-based access control. In: Proceedings of the 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003). (2003) 43–50
22. Power, D.J., Politou, E.A., Slaymaker, M.A., Simpson, A.C.: Towards secure grid-enabled healthcare. Software: Practice and Experience **35**(9) (2005) 857–871
23. Hosmer, H.H.: Metapolicies I. ACM SIGSAC Review **10**(2–3) (1992) 18–43
24. Spivey, J.M.: The Z Notation: A Reference Manual. Prentice Hall (1992)
25. Woodcock, J.C.P., Davies, J.W.M.: Using Z: Specification, Refinement, and Proof. Prentice-Hall (1996)