Soutei, a Logic-Based Trust-Management System
System Description

Andrew Pimlott[1] and Oleg Kiselyov[2]

[1] Planning Systems, Inc., Slidell, LA
andrew@pimlott.net
[2] Fleet Numerical Meteorology and Oceanography Center, Monterey, CA
oleg@pobox.com

Abstract. We describe the design and implementation of a trust-management system Soutei, a dialect of Binder, for access control in distributed systems. Soutei policies and credentials are written in a declarative logic-based security language and thus constitute distributed logic programs. Soutei policies are modular, concise, and readable. They support policy verification, and, despite the simplicity of the language, express role- and attribute-based access control lists, and conditional delegation.

We describe the real-world deployment of Soutei into a publish-subscribe web service with distributed and compartmentalized administration, emphasizing the often overlooked aspect of authorizing the creation of resources and the corresponding policies.

Soutei brings Binder from a research prototype into the real world. Supporting large, truly distributed policies required non-trivial changes to Binder, in particular mode-restriction and goal-directed top-down evaluation. To improve the robustness of our evaluator, we describe a fair and terminating backtracking algorithm.

Keywords: Access control, Security Language, Logic Programming, Datalog, Non-determinism, Backtracking, Haskell.

1 Introduction

An authorization system is called upon to advise whether an access request, such as fetching a web page, should be permitted. To reach a decision, the system consults attributes of the request (such as the users and resources involved), information about the world, and policies. While simple data structures like access control lists suffice as policies in some cases, they fail to accommodate large and volatile sets of users and resources, complicated constraints, and distributed administration.

Blaze et al. [3] introduced the trust-management system as a standard interface for applications (web servers, etc.) to ask an authorization system whether a requested action should be allowed, and a standard language for writing distributed and interoperable policies. Trust management is characterized by the following design principles, which have proved to be both flexible and efficiently implementable:

– The application provides facts about the requested action (axioms);
– policies specify the derivation of other facts (inference rules);

- the authorization system is a policy compliance checker, which advises the application whether an application-defined action (formula) should be allowed (can be inferred from the axioms and inference rules);
- the compliance checker is a deduction engine.

These principles have been validated by KeyNote [2], the most well-known trust-management system with several real-world applications. The security language of KeyNote however has several notable drawbacks [15, 13]: It is impossible in general to analyze the effect of KeyNote assertions, for example to verify global policy constraints.

We argue that a trust-management system should be based upon a single, declarative language, with transparent semantics and a minimum of primitives. The language should be easy to read, write, and modify. It should express complicated policies without redundancy, and allow independent policies to be composed. It should be amenable to automated formal analysis and efficient, robust, high-assurance implementation.

Experience with a common class of policies, firewall configurations, illustrates the hazards when these requirements are not met. Firewall rules are numerous, and their overall effect is difficult to understand; a seemingly small change may lead to a cascade failure [19], which, absent policy verification, may go undetected. A study of firewall configuration errors [23] found flaws in every firewall examined: “Only one of the 37 firewalls exhibited just a single misconfiguration. All the others could have been easily penetrated by both unsophisticated attackers and mindless automatic worms.”

DeTreville [7] introduced a logic-based security language Binder for expressing policies in distributed systems. Binder is an extension of the logic programming language Datalog, which, in turn, is a subset of pure Prolog. Despite having few primitive constructs (only one distinguished relation `says`), Binder is shown to be more expressive than most existing security languages. The ease of naming and referring to other policies via `says` is especially attractive. Despite its power, Binder is concise and comprehensible: A statement in Binder can be read as a declarative, stand-alone English sentence [7]. Binder has a clean design and sound logical foundations [1]. In our view, Binder meets the criteria for a trust-management language put forth above. Alas, it was just a research prototype with no implementation.

Soutei can be viewed as KeyNote whose security language is replaced with Binder—or as the first practical implementation of Binder. However in producing a working system, we had to specify many aspects left undefined in DeTreville’s paper, and add several extensions. Many of the extensions, such as the interaction between an application and Soutei, are notably influenced by KeyNote.

Our contributions are:

- An extension of Binder to specify the interaction between the authorization system and the application. We introduce a dedicated `application` namespace for the application to supply attributes of the request and environment data, which can be used in policies.
- A re-interpretation of the distinguished Binder relation `says`, allowing us to use a goal-directed resolution-based strategy, which can cope with the large number of distributed policies.

- A mode system for assertions, with mode-restricted forms and static checking of the satisfaction of the restrictions.
- A general-purpose efficient backtracking algorithm with fairness and termination properties.
- Two implementations, designed to support various applications.[1]
- Deployment of Soutei into a publish-subscribe web service with distributed and compartmentalized administration. Administrative actions such as the creation of resources and the attachment of authorization policies are themselves subject to authorization.

The next section introduces Soutei by a series of examples. Section 3 presents a real-world use case, using Soutei with a publish-subscribe web service. In Section 4, we describe three aspects of Soutei implementation: its goal-directed resolution-based evaluation, modes, and backtracking algorithm. These aspects represent notable deviations from Binder, influenced by practical requirements, and can be re-used in other similar systems. We present performance measurements in Section 5. Section 6 examines related work; and Section 7 concludes. The formal semantics of Soutei is given in the Appendix.

2 Soutei by Example

We present the fundamentals of Soutei with a simple policy:

```
may(read) :- application says ip-address(?IP),
             internal(?IP).
internal(#p10.10.1.1). ; trusted internal clients
internal(#p10.10.1.2).
```

The syntax of Soutei is taken mainly from Binder and Datalog; however, there are minor departures: A leading question mark rather than a capital letter identifies logical variables, to avoid confusion with proper names. Atomic constants include symbols (which may be enclosed in double quotes, to escape spaces and similar characters) and IP addresses. A semicolon begins a single-line comment.

The semantics of Soutei is likewise close to Binder and Datalog [7]. The above example consists of three clauses: an inference rule for the predicate `may`, and two axioms for the predicate `internal`. The `may` rule has a body consisting of two atoms, both of which must be satisfied to derive `may(read)`. The `internal` predicate has two clauses, either of which may satisfy `ip-address(?IP)`. Informally, the policy can be read, "read access is granted to clients with internal IP addresses, which are 10.10.1.1 and 10.10.1.2".

The interpretation of the predicate `may` is *not* built-in, but given by the application requesting advice, which presents the formula for Soutei to derive. In our example, an application would submit formulas such as `may(read)` and `may(write)`. The application also sends additional axioms representing facts about the request; in this

[1] A simple web interface to Soutei is available at `http://www.metnet.navy.mil/cgi-bin/mcsrvr/soutei-demonstration`.

case, we expect it to send an `ip-address` axiom. Again, the application, not Soutei, chooses the predicates. These axioms, along with the formula to prove, are analogous to an action in KeyNote.

We have created an s-expression-based protocol for applications to request advice from Soutei. The application in this example might open a TCP connection to a Soutei server and send

```
(req-17 query (may read) (ip-address #p10.10.1.1) ...)
```

Soutei responds with a boolean result, indicating whether it could derive the formula: `(req-17 #t)`.[2] It is the responsibility of the application to ensure that Soutei's advice is followed. In RFC 3198 [22] terms, Soutei acts as a Policy Decision Point, and the application as a Policy Enforcement Point.

We have left to explain the primitive form `says`. As in Binder, the policy in Soutei is a set of (distributed) *assertions*, named collections of facts and rules. The distinguished assertion `system`, analogous to the `POLICY` principal in KeyNote, represents the top-level policy, and is loaded from a start-up configuration file. Derivation of the formula presented by the application is always attempted within the `system` assertion.[3] The distinguished assertion `application` contains the facts sent by the application (as well as some some built-in predicates that cannot be defined within Soutei; see Section 4.2). The form `says` indicates that a predicate should be resolved within another assertion, allowing assertions to refer to each other. In our example, the `ip-address` predicate is to be resolved within the `application` assertion. The semantics of `says` is discussed further in Section 4.1.

Assertions (other than `system` and `application`) may come from various sources, depending on the system architecture. In our current implementation, they are pushed to Soutei; however, they might instead be stored in a local, remote, or distributed database, pulled from a subscription service, or be delivered with a decision request as signed credentials. Because an assertion has no force unless referred to by `says`, we can allow every user to control his own assertion without worrying about compromising the policy.

Let us consider some more sophisticated examples. We can model access control lists, statements enumerating the users and roles who have access to given resources. The user and resource data are, like the IP address, passed as application facts.

```
may(?access) :- application says resource(?resource),
                application says public-key(?key),
                user-key(?user, ?key),
                role-member(?user, ?role),
                acl-may(?access, ?resource, ?role).
; users and their public keys
user-key(Peter, "rsa:Z2FuZ3N0YQ==").
user-key(Bill, "rsa:eWVhaCBoaQ==").
; roles
role-member(Peter, programmer).
```

[2] `req-17` is a request identifier chosen by the client; it is not interpreted Soutei.

[3] Our example policy was tacitly the `system` assertion.

```
role-member(Bill, manager).
; ACLs
acl-may(read, TPS-report-memo, programmer).
acl-may(read, TPS-report-memo, manager).
acl-may(write, TPS-report-memo, manager).
```

Following KeyNote, we identify users by their public keys, which may be delivered to the application in an SSL handshake. As public keys are unwieldy, we create convenient aliases with `user-key`. The predicate `role-member` expresses role membership, and `acl-may` lists the permissions of different roles on resources.

Storing role membership, ACLs and associations of user names with their keys in the top-level policy is generally inappropriate. The `system` policy is modifiable only by the Soutei administrator, but (hypothetically) the human resources (HR) department should maintain users' keys, the application owner should manage roles, and resource owners should control ACLs. We can divide authority in this way using `says`.

```
may(?access) :- application says resource(?resource),
                application says resource-owner(?owner),
                application says public-key(?key),
                hr says user-key(?user, ?key),
                app-owner says role-member(?user, ?role),
                ?owner says acl-may(?access, ?resource, ?role).
```

The assertions of HR, the application owner, and the resource owner are now all involved in the authorization decision, although each has a different and limited role. These assertions may use the full Soutei language; in particular, they may use `says` to further delegate their authority.

Another use of `says` is to reconcile policies, even policies using different vocabularies.

```
may(?access) :- authority1 says can(?access)
                application says resource(?resource),
                authority2 says ok(?access, ?resource).
may(read) :- authority1 says can(read).
```

In this example, `authority1` alone may grant read access, but other access requires the agreement of both authorities. Note it is not a problem that the authorities use different predicates.

The Soutei documentation [13] describes the use of Soutei for role-based and capability-based access control, discusses monotonicity, revocation, blacklisting, and introduces additional use cases (which we use as the regression test of our implementation).

3 A Real-Life Use Case

We have integrated Soutei with a general purpose publish-subscribe data distribution web service, Metcast Channels [12]. Clients of this service can publish data into a channel, monitor a channel for data, and request the list of all channels. Metcast Channels have been used in an operational environment for several years. Previously, Metcast Channels used simple access control lists, which were quite inflexible. Assistance of the administrator was required to install or modify access control lists of any channel. As the system became more widely used and the number of user communities and their channels increased, so did the load on the administrator. Ideally the administrator should set up the top-level policy and leave managing particular channels to their owners.

Now, Metcast Channels is a Soutei application. Upon receiving a request to read, write, create, or delete a channel, the server seeks the advice of Soutei. Users may also submit their own Soutei assertions, via a special channel. A submitted assertion is associated with the user's authenticated name and stored in a local database. Since they pass through a channel, policy changes are authorized by Soutei, though we currently allow any authenticated user to submit an assertion. The following scenario demonstrates how this system overcomes the previous difficulties, and provides additional benefits.

The `system` policy is quite minimal: it does not grant privileges itself, it only delegates them. Specifically, it delegates permission to administer channels to a system administrator; permission to read and write channels to the channel owner; and all permissions to a security officer, for use in emergencies. In order to emphasize the distinction between channel administration and channel access operations (and to guard against carelessness), Metcast Channels uses two predicates, `may-admin` and `may`.

```
; delegate channel administration to sam.sysadmin
may-admin(?access) :- sam.sysadmin says may-admin(?access).
; delegate channel access to the channel owner
may(?access) :- application says channel-owner(?owner),
                ?owner says may(?access).
; delegate all access to the security officer
may-admin(?access) :- ed.emergency says may-admin(?access).
may(?access)       :- ed.emergency says may(?access).
```

The security officer's assertion will normally be empty, so access will only be granted by the first two rules; however when the security officer does submit an assertion, it may bypass the usual rules.

A user, `cam.create`, wishes to create a channel, in order to publish his blog. In some systems, create access is tricky because there is no existing resource on which to hang the permissions. Soutei does not have a rigid vocabulary, though, so we can express an action not involving a resource naturally. On `cam.create`'s request, `sam.sysadmin` submits the rule

```
may-admin(create) :- application says user(cam.create).
```

The user `cam.create` then creates the channel `CamsBlog`. Note that by the very creation of this channel (and without administrator intervention), `cam.create`'s assertion comes into force to control access. He submits the following policy, granting himself read and write permissions, and authorizing `don.delegate` to grant read permission—but only within the computer science department.

```
may(?access) :- application says channel(CamsBlog),
                application says user(cam.create),
                known-access(?access).
```

```
known-access(read).
known-access(write).
may(read) :- application says channel(CamsBlog),
             application says user-department(CS),
             don.delegate says may(read).
```

The ability to delegate, in a controlled way, represents a major advance over access control lists.

`don.delegate` is careless. His policy is:

```
may(read) :- application says channel(CamsBlog).
```

This permissive rule appears to grant read access to everyone. Of course, it does not: the chain of proof stops in `cam.create`'s policy if the user is outside the CS department; `don.delegate`'s assertion is never considered.

The stipulation of `cam.create` can, however, be circumvented by `ed.emergency`, because his authority comes directly from the `system` assertion. In a situation where access to information is paramount, he submits the policy

```
may(read).
```

Anyone can now read all channels. In the defense community, this flexibility is called Risk Adaptable Access Control (RAdAC), and is considered a critical requirement for the Global Information Grid [5].

4 Implementation

We initially implemented Soutei in Scheme, using the logical programming system Kanren [11]. We then re-implemented Soutei in Haskell. Decisive factors in this decision were static typing, manifest distinction between pure and effectful code, closeness to the specification language, and QuickCheck property testing. These all help increase one's confidence in the code, which is of special importance to us since Soutei is intended for high-assurance applications.

In this section, we discuss three particular aspects: goal-directed resolution-based evaluation, mode analysis, and a fair and terminating backtracking algorithm. These aspects represent advances over Binder and illustrate how practical needs for scalability, robustness, and error reporting influence design decisions.

4.1 Goal-Directed Resolution-Based Evaluation

Binder implicitly assumed that all assertions are immediately available. The relation `says` was taken to qualify predicates with assertion names. The formal qualification rules are given in [7] and logically justified in [1]. Once all predicates become qualified, one may regard the qualifier as an extra argument to the predicate, and then treat the collection of all available assertions as an ordinary, non-distributed Datalog program, to be evaluated with the conventional bottom-up strategy.

We assume a potentially large number of assertions, which makes keeping them all in memory impractical. We further assume that not all assertions are locally known—the logical program of Soutei is truly distributed. Under these conditions, whole-program translation into Datalog is not feasible. Goal-directed resolution-based evaluation seems more suitable, as it consults only those assertions that may actually be needed for the goal at hand. In fact, Soutei needs only one proof to advise access, so it may consult fewer.

We therefore view each assertion as a set of clauses [17], and `a says B` as a judgement that there is a proof of formula B given the clauses in assertion `a`. When such a judgement is required—and only then—we load the assertion and proceed with resolution. We use the backchaining rule to prove a formula given the clauses in an assertion [17, 4]. Appendix A shows the formal semantics of Soutei in full.

Our Haskell implementation follows the established pattern [18, 6] of embedding a resolution-based logical programming system in Haskell. To apply the backchaining rule, we non-deterministically choose a clause for the target predicate (see Section 4.3), unify with the head to build up the substitution, then sequentially resolve the body.

4.2 Modes

The distributed nature of Soutei motivates another aspect of the design. In Binder, one may use `says` whose left argument, the assertion name, is a logical variable—even an uninstantiated variable. This would have the effect of enumerating all assertions, which is fine when all assertions are locally available, but impractical in a distributed system. Therefore, we disallow this situation: the first argument of `says` must be instantiated, that is, it must have the `In` mode [20]. The `application` namespace has several built-in predicates that are similarly mode-restricted, such as `ip-of`, for matching an IP address against a network range. It expects both arguments to be instantiated, so they have the `In` mode.

Mode enforcement can be performed as a run-time instantiatedness check, whose failure either triggers an error or suspends the evaluation of a goal. But debugging these cases—which could arise from the interaction of many assertions—would be painful, and one could never be sure they are all eliminated. We prefer to detect and report ill-moded assertions when they are submitted. Therefore, we have designed and implemented a static mode analysis, which ensures that logical variables are used with the correct modes.

The static mode analysis poses several challenges. For ease of use, we would like to avoid explicit mode declarations. On the other hand, the distributed nature of Soutei makes whole-program mode analysis all but impossible. Fortunately, we can impose a simplifying and yet not-too-limiting restriction. For any predicate defined in an assertion, we require that all of its arguments be instantiated when the predicate succeeds. This requirement is tantamount to an implicit `Out` mode declaration for all arguments. We reject an assertion at submission time if it is not well-moded with respect to the implicit mode declaration.

The `application` assertion, which consists of application facts and built-in predicates, is treated specially. Facts trivially satisfy the `Out` mode; however the built-in predicates may have other modes, which must be known to the mode checker.

The intuition for the mode checker is that after a logical variable is used with an `Out` mode, it is safe to use in mode-restricted contexts. More precisely, we first assume that all atoms in the body of a rule, except those with mode-restricted `application` predicates, instantiate their logical variable arguments when they succeed. We then attempt to demonstrate two safety conditions: that all mode-restricted arguments to `application` predicates and to `says` are instantiated by a previous body atom; and that all logical variables in the head of the rule are instantiated somewhere in the body, that is, they satisfy the implicit `Out` mode. If so, the rule is well-moded. We leave a full elaboration of the mode inference judgments, and a proof of soundness, for future work.

We can apply this procedure to the rule

```
may(?access) :- application says ip-address(?IP),
                application says ip-of(?IP, #n192.168.0.0/8),
                administrator(?admin),
                ?admin says may(?access).
```

The logical variable `?IP` is instantiated by the non-mode-restricted `application` predicate `ip-address` before it is used in the mode-restricted `ip-of`. The logical variable `?admin` is instantiated by the predicate `administrator` before it is used to the left of `says`. And the head variable `?access` is instantiated by the body predicate `may`. So the rule passes the mode checker. If the last two body atoms were reversed, so that `?admin` were used with `says` before being instantiated, the rule would be rejected with a mode error.[4]

The requirement that predicates have the `Out` mode has a practical impact on their design. To give full access to a super-user, we cannot write

```
may(?access) :- application says user(?user),
                super-user(?user).
```

If `may` were called with `?access` uninstantiated, it would remain uninstantiated, violating the implicit `Out` mode declaration.

Rather, we write the assertion as[5]

```
may(?access) :- application says user(?user),
                super-user(?user),
                known-access(?access).
known-access(read).
known-access(write).
```

We find that this style, while more verbose, is clearer about what is being granted. It will also give more information to a future policy verifier.

For similar reasons, we model users and resources as application facts, rather than (following DeTreville) as arguments to the formula to prove. Consider the proposed super-user rule

[4] We are investigating allowing the mode checker to reorder atoms in order to satisfy mode restrictions, as in Mercury [20].

[5] This explains the use of `known-access` in Section 3.

```
may(?user, ?access, ?resource) :- super-user(?user),
                                  known-access(?access).
```

The mode checker rejects this rule, as the `?resource` argument does not satisfy the `Out` mode. To repair it, we would need a predicate enumerating all resources, which is often infeasible. Our style sidesteps the issue.

4.3 Fair and Terminating Backtracking Algorithm

The goal-directed evaluation of Section 4.1 relied upon backchaining, which involves non-deterministic selection of the clause to apply. In this section we describe a backtracking algorithm implementing that non-deterministic choice. Although expressed as a Haskell monad, it can be implemented in other languages (and has been, in OCaml and Scheme). We use Haskell as a typed, executable specification language.

Following Wadler [21], we realize non-deterministic computations in Haskell as computations in a particular monad, `MonadPlus`. Specifically, the type class `MonadPlus` defines the interface for the minimal set of operations to express non-deterministic computations: `mzero` creates a failing computation; `return a` creates a deterministic computation that yields `a`; `m >>= k` sequences two computations, passing the results of computation `m` to `k`; and `mplus m1 m2` represents choice between computations `m1` and `m2`. There is also an operation to run the computation, obtaining its answer (if any).

Wadler also introduced the list monad, an implementation of the `MonadPlus` interface that amounts to depth-first search. Although the most common `MonadPlus` implementation, and part of Haskell98 standard, we cannot use it for Soutei if we intend to be robust against careless or malicious policies. Consider the assertion:

```
loop(?x) :- loop(?x).
may(read) :- loop(1).
may(read).
```

Attempting to prove `may(read)` with depth-first backchaining, and selecting clauses in the order given, will diverge trying to prove `loop(1)`. We will never get to the second `may` clause, so access will not be granted, and—worse—Soutei will enter an infinite loop. If `loop` were in a separate assertion, referred to by `says`, that assertion could effect a denial of service attack.

Seres and Spivey [18] suggest using breadth-first search instead, and give a monad implementing this strategy. Breadth-first search is complete: it always finds a solution, provided one exists. However, it has large memory requirements, to hold the entire frontier of the computation, and so is not practical.

Depth-first search with iterative deepening brings completeness to depth-first search, at the cost of repeating parts of the computation, including side-effects such as loading assertions. We prefer to avoid this duplicate work.

Kiselyov, Shan, Friedman, and Sabry [14] present a non-determinism monad with fair operations for combining computations that may make an infinite number of choices. Alas, these operations are of no help in dealing with computations, such as our proof of `loop(1)`, that never yield an answer.

```
import Control.Monad

data FStream a = Nil | One a | Choice a (FStream a)
               | Incomplete (FStream a)

instance Monad FStream where
    return = One

    Nil          >>= f = Nil
    One a        >>= f = f a
    Choice a r   >>= f = f a `mplus` (Incomplete (r >>= f))
    Incomplete i >>= f = Incomplete (i >>= f)

instance MonadPlus FStream where
    mzero = Nil

    mplus Nil r'               = Incomplete r'
    mplus (One a) r'           = Choice a r'
    mplus (Choice a r) r'      = Choice a (mplus r' r)   -- interleave
    mplus r@(Incomplete i) r' =
      case r' of                                          -- try alternative
              Nil          -> r
              One b        -> Choice b i
              Choice b r'  -> Choice b (mplus i r')
              Incomplete j -> Incomplete (mplus i j)

yield :: FStream a -> FStream a
yield = Incomplete

runFS :: Maybe Int -> FStream a -> [a]
runFS _ Nil = []
runFS _ (One a) = [a]
runFS n (Choice a r) = a : (runFS n r)
runFS (Just 0) (Incomplete r) = []                       -- exhausted cost
runFS n (Incomplete r) = runFS n' r
    where n' = liftM pred n
```

Fig. 1. Complete FStream implementation

To overcome non-termination in such cases, we introduce a search algorithm that enjoys fairness, termination, and efficiency properties not found in prior work. Figure 1 shows the *complete* implementation.[6] We represent non-deterministic computations by terms of the algebraic type `FStream a`. In addition to the conventional terms `Nil` and `Choice` (analogous to `[]` and `(:)` in the list monad) representing failure and a choice [10], we add a term `One a` for a deterministic computation, and a term `Incomplete (FStream a)` denoting a suspended computation. The function `runFS` runs the computation.

The introduction of `Incomplete` allows us to capture the continuation of a computation—even a divergent one. Denotationally, `Incomplete` is the identity on

[6] We have also implemented the algorithm as a monad transformer, which allows us to execute `IO` operations, such as fetching a remote assertion, during computation.

computations. Operationally, it is a hint to the evaluator to explore alternative computations, if any. The definition of mplus honors the hint by deferring a computation marked with Incomplete; it also adopts the interleaving technique from [14] to achieve fairness. The result is a balance between depth- and breadth-first search strategies that performs well in practice. We leave a complete analysis of FStream for future work.

The programmer must apply the Incomplete hint (using the exported function yield) to potentially divergent computations. In Soutei, we call yield every time we choose a clause in the backchaining rule. Therefore, even a tight recursion such as proving loop(1) can be interrupted to try the second may clause. We believe that our use of FStream makes Soutei's resolution algorithm complete.

A final problem remains, of a top-level computation that neither fails nor succeeds in any evaluation strategy, such as our example with the second may clause removed. We require a way of giving up a fruitless search. Here Incomplete provides a separate benefit, as a measurement of the cost of a query. We take advantage of this in the top-level runFS. When runFS eliminates an Incomplete, it decrements a counter, allowing the programmer to control how long the computation should run.

Soutei passes a number chosen by the Soutei administrator to runFS, and of course does not advise access if the cost is consumed before finding a proof. We emphasize that although Soutei may fail to find an existing proof in this case, our logic itself is decidable, as can be seen from the translation to Datalog in Section 4.1. Further, while Datalog is decidable in polynomial time with a bottom-up strategy, the large number of possible assertions would make running this algorithm to completion impractical for providing timely authorization advice; a timeout would still be needed. Thus, the effectiveness of Soutei is not fundamentally hampered by our choice of evaluation strategy.

As a demonstration of the power of FStream, Soutei is able to deal with left-recursive and divergent rules, such as this search for paths in a directed graph:

```
path(?x, ?y) :- path(?x, ?z), edge(?z, ?y).
path(?x, ?y) :- edge(?x, ?y).
edge(1, 2).
edge(2, 1).
edge(2, 3).
```

The graph contains a loop between points 1 and 2. Further, the path predicate is left-recursive, so a depth-first evaluation strategy would loop without even deriving path(1, 2). However, Soutei avoids these pitfalls and can derive path(1, 3).

5 Performance Measurements

We use the scenario in Section 3 as the primary benchmark of our implementations. When used with Metcast Channels, the overhead added by Soutei was within measurement error. Therefore, we extracted the Soutei requests performed by Metcast Channels during the scenario, five new assertions and 17 queries, into a script that calls Soutei directly.

We ran this script against the Haskell implementation of Soutei running as a TCP server. As an average over many repetitions, the scenario took 55 ms. (This and all other measurements were performed on a 1.8 GHz Pentium 4 PC with 1 GB RAM. Programs were compiled by the Glasgow Haskell Compiler, version 6.4.1, with the `-O2` flag.)

This test included the overhead of network communication, protocol processing, and saving assertions to disk. To eliminate this, we added a mode to the script to call Soutei in-process. In this mode, the scenario took only 5.9 ms. We suspect that the overhead in the server version of Soutei could be trimmed; however since its performance is adequate, we have not attempted this.

Both tests placed a large weight on submission of assertions. A new assertion requires parsing, mode checking, and transformation into a form optimized for query execution. In real use, queries would likely be far more common than policy changes, so we modified the script to submit the assertions once and run the queries many times. This gave us a measurement of just the logic engine. The 17 queries in the scenario took 2.7 ms, or .16 ms per query.

The total virtual memory required by these tests was around 4-5 MB, as reported by the operating system; however, some of this can be accounted to the Haskell garbage collector.

6 Related Work

Because of its obvious importance, the area of access control has been the subject of much research and development. Unfortunately, many deployed access control systems (e.g., firewall access control lists and UNIX permissions) are ad hoc and inexpressive. Trust-management systems [3, 2] introduced a principled approach and a security language. Alas, the design of an expressive, tractable and logically sound security language turns out to be quite hard: Li and Mitchell [15] show that the language in KeyNote has features that impede analysis, and Abadi [1] points out other languages and their shortcomings. The same paper describes the logical justification for Binder. It seems to us that Binder offers the best balance of simplicity and expressivity.

Li and Mitchell [15] investigate constrained Datalog in trust-management systems, prove the tractability of important domains such as trees and ranges, and describe the application to a particular trust-management system RT. We believe Binder offers a better modularity, via `says`. However, adding constraints to Soutei would increase its expressiveness and remains an interesting research direction.

Garg and Pfenning [9] introduce a constructive logic for authorization, and prove non-interference theorems: for instance, only users mentioned directly or indirectly by the `system` assertion can affect decisions. Their logic appears quite similar to Soutei, and we are investigating whether their techniques can be adopted for off-line policy verification.

XACML (eXtensible Access Control Markup Language) 2.0 [24] is an OASIS standard for authorization policies. It is a large specification, and contains some of the same features, such as arithmetic functions, that reduce the tractability of KeyNote; yet XACML cannot express delegation. Fisler et al. [8] have verified XACML policies using decision diagrams, but their work is limited to a subset of the language, and faces scalability questions. They acknowledge that certain XACML constructs "fall outside the scope of static validation."

Backtracking transformers are derived in [10] and [14]. Only the second paper deals with fairness and neither can handle left-recursion.

7 Conclusions and Future Work

We have presented a trust-management system Soutei that combines the strengths of KeyNote and Binder and is a practical alternative to various ad hoc user-, role-, and attribute-based access control lists. We inherit from Binder distributed policies, simplicity, and independence of any application vocabulary. The system is implemented and can be used with any application.

We have validated the system by integrating it with a publish-subscription web service. With the current (quite small) number of assertions, the impact of Soutei on the performance of the system is negligible.

The practical requirements of scalability to many, possible remote assertions motivated goal-directed resolution-based evaluation. To implement this robustly, we have designed a novel fair and terminating backtracking algorithm. To ensure statically that all assertions will not have to be loaded to resolve a query, we have designed a mode analysis for assertions, which places only a light burden on policy authors.

We are currently developing policy administration user interfaces and off-line policy verification tools, and integrating Soutei into additional applications.

Acknowledgements. We thank Paul Steckler, Frank Pfenning, Christopher Thorpe, and the anonymous reviewers for many helpful suggestions.

References

1. ABADI, M. Logic in access control. In LICS [16], pp. 228–233.
2. BLAZE, M. Using the KeyNote Trust Management System. `http://www.crypto.com/trustmgt/`, Mar. 2001.
3. BLAZE, M., FEIGENBAUM, J., AND LACY, J. Decentralized trust management. In *IEEE Symposium on Security and Privacy* (May 1996).
4. BRUSCOLI, P., AND GUGLIELMI, A. A tutorial on proof theoretic foundations of logic programming. In *ICLP* (2003), C. Palamidessi, Ed., vol. 2916 of *Lecture Notes in Computer Science*, Springer, pp. 109–127.
5. CHISHOLM, P. IA roadmap. *Military Information Technology 9*, 5 (25 July 2005).
6. CLAESSEN, K., AND LJUNGLÖF, P. Typed logical variables in haskell. *Electr. Notes Theor. Comput. Sci. 41*, 1 (2000).
7. DETREVILLE, J. Binder, a logic-based security language. In *IEEE Symposium on Security and Privacy* (2002), pp. 105–113.
8. FISLER, K., KRISHNAMURTHI, S., MEYEROVICH, L. A., AND TSCHANTZ, M. C. Verification and change impact analysis of access-control policies. In *International Conference on Software Engineering* (May 2005).
9. GARG, D., AND PFENNING, F. Non-interference in constructive authorization logic. Submitted for publication, Oct. 2005.
10. HINZE, R. Deriving backtracking monad transformers. In *ICFP '00: Proceedings of the 5th ACM SIGPLAN International Conference on Functional Programming* (2000), ACM Press, pp. 186–197.

11. A declarative applicative logic programming system. `http://kanren.sourceforge.net/`, 2005.
12. KISELYOV, O. Metcast Channels. `http://www.metnet.navy.mil/Metcast/Metcast-Channels.html`, Feb. 2003. The working server with Soutei Authorization can be accessed via `http://www.metnet.navy.mil/cgi-bin/oleg/server`.
13. KISELYOV, O. Soutei: syntax, semantics, and use cases. `http://www.metnet.navy.mil/Metcast/Auth-use-cases.html`, 13 June 2005.
14. KISELYOV, O., SHAN, C., FRIEDMAN, D. P., AND SABRY, A. Backtracking, interleaving, and terminating monad transformers. In *ICFP '05: ACM SIGPLAN International Conference on Functional Programming* (2005), ACM Press.
15. LI, N., AND MITCHELL, J. C. Datalog with constraints: A foundation for trust management languages. In *PADL* (2003), V. Dahl and P. Wadler, Eds., vol. 2562 of *Lecture Notes in Computer Science*, Springer, pp. 58–73.
16. *18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003, Ottawa, Canada, Proceedings* (2003), IEEE Computer Society.
17. MILLER, D., AND TIU, A. F. A proof theory for generic judgments: An extended abstract. In LICS [16], pp. 118–127.
18. SERES, S., AND SPIVEY, J. M. Embedding Prolog in Haskell. In *Proceedings of the 1999 Haskell Workshop* (1999), E. Meier, Ed., Tech. Rep. UU-CS-1999-28, Department of Computer Science, Utrecht University.
19. SINGER, A. Life without firewalls. *USENIX ;login: 28*, 6 (Dec. 2003), 34–41.
20. SOMOGYI, Z., HENDERSON, F., AND CONWAY, T. The execution algorithm of Mercury, an efficient purely declarative logic programming language. *J. Log. Program. 29*, 1-3 (1996), 17–64.
21. WADLER, P. How to replace failure by a list of successes: A method for exception handling, backtracking, and pattern matching in lazy functional languages. In *FPCA* (1985), pp. 113–128.
22. WESTERINEN, A., SCHNIZLEIN, J., STRASSNER, J., SCHERLING, M., QUINN, B., HERZOG, S., HUYNH, A., CARLSON, M., PERRY, J., AND WALDBUSSER, S. Terminology for policy-based management. RFC 3198, Nov. 2001.
23. WOOL, A. A quantitative study of firewall configuration errors. *IEEE Computer 37*, 6 (2004), 62–67.
24. OASIS eXtensible Access Control Markup Language (XACML). Version 2.0. `http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml`, Feb. 2005.

A The Formal Semantics of Soutei

Table 1 describes syntax and meta-variables used in this section. Predicate constants have the implied arity $n \geq 1$. Logic variables range over constants. A substitution is a finite map from logic variables to terms, mapping no variable to itself. An application of a substitution is written in postfix notation.

In an assertion clause, H is an unqualified formula, $\overline{B}$ is a set of zero or more general formulas, $\{y_1 \cdots y_m\}, m \geq 0$ is a set of variables free in $\overline{B}$ but not in H, $\{x_1 \cdots x_n\}, n \geq 0$ is a set of variables that are free in H (they are all free in $\overline{B}$, too); symbol $\overset{\Delta}{=}$ is just a separator.

Table 1. Syntax and meta-variables

Constants	$=$	$a, b,$ `application`, `system`
Predicate constants	$=$	p, q
Logic variables	$=$	x, y, z
Terms t	$::=$	$a \mid x$
Substitutions	$=$	θ, ρ
Unqualified formulas A, H	$::=$	$p\, t_1 \ldots t_n$
Qualified formulas Q	$::=$	t `says` A
General formulas B	$::=$	$A \mid Q$
Assertion clause c	$::=$	$\forall x_1 \cdots \forall x_n\ \exists y_1 \cdots \exists y_m\ H \triangleq \overline{B}$
Assertion Γ	$::=$	$\{c, \ldots\}$

Table 2. Judgments

$a \vdash_n \Gamma$	a is the name of the assertion Γ
$\Gamma \vdash B$	Formula B holds in the assertion Γ
$a \vdash B$	Formula B holds in the assertion named a

Table 3. Semantics

$$\frac{b \vdash A}{\Gamma \vdash b \ \texttt{says}\ A}\ (\mathit{says})$$

$$\frac{a \vdash_n \Gamma \quad \Gamma \vdash A}{a \vdash A}\ (\mathit{lookup})$$

$$\frac{\{\Gamma \vdash B\theta \mid B \in \overline{B}\} \quad \forall x_1 \cdots \forall x_n\ \exists y_1 \cdots \exists y_m\ H \triangleq \overline{B} \in \Gamma \quad A = H\theta}{\Gamma \vdash A}\ (\mathit{backchain})$$

$$\frac{}{\texttt{application} \vdash \mathrm{neq}\ a\ b}\ (\mathit{neq}) \qquad \text{constants } a \text{ and } b \text{ are distinct}$$

$$\frac{}{\texttt{application} \vdash \mathrm{ip\text{-}of}\ a\ b}\ (\mathit{ip}) \qquad a, b \text{ represent IP network addresses and } a \text{ is included in } b$$

Table 2 lists the judgements. Soutei accepts a formula A as input and replies whether there exists a substitution ρ such that the judgement `system` $\vdash A\rho$ can be deduced according to the rules in Table 3. The latter table omits the axioms for $a \vdash_n \Gamma$ judgements: These axioms specify the contents of `system`, `application`, and other assertions; the set of such axioms is specific to a particular invocation of Soutei.

The rule *(says)* is similar to the conventional modality law: if $b \vdash A$ holds, then $\Gamma \vdash b$ `says` A holds in any any *existing* assertion Γ [1]. In the rule *(backchain)* the set of premises of the rule is given explicitly as a set, which may be empty. The domain of the substitution θ includes both $\{x_1 \cdots x_n\}$ and $\{y_1 \cdots y_m\}$.