Efficient Trust Management Policy Analysis from Rules [*]

Katia Hristova [†], K. Tuncay Tekle, and Yanhong A. Liu

Computer Science Department
State University of New York
Stony Brook, NY 11794
{katia,tuncay,liu}@cs.sunysb.edu

Abstract

This paper describes a systematic method for deriving efficient algorithms and precise time complexities from extended Datalog rules as it is applied to the analysis of trust management policies specified in SPKI/SDSI, a well-known trust management framework designed to facilitate the development of secure and scalable distributed computing systems. The approach of expressing policy analysis problems as extended Datalog rules is much simpler than previous techniques for analysis of SPKI/SDSI policies. Our method also derives better, more precise time complexities than before in addition to generating complete algorithms and data structures. The method is general, with many applications beyond policy analysis. It extends our previous method for Datalog to handle list constructors, external functions, and queries.

Categories and Subject Descriptors D.3 [*Programming Languages*]: Processors—code generation, optimization; D.3.4 [*Programming Languages*]: Language Classifications—constraint and logic languages; D.4.6 [*Operating Systems*]: Security and Protection—access controls; E.1 [*Data*]: Data Structures—arrays, lists, queues, records; F.2 [*Analysis of Algorithms and Problems Complexity*]: Nonnumerical Algorithms and Problems—computations on discrete structures; I.2.2 [*Artificial Intelligence*]: Automatic Programming—Program transformation

General Terms security, algorithms, languages, performance

Keywords access control, policy analysis, security, algorithm, time complexity

1. Introduction

Trust management is a unified approach to specifying and enforcing security policies in distributed systems [11, 5] and has become increasingly important as systems become increasingly interconnected. At the same time, logic-based languages and frameworks have been used increasingly for expressing security and trust management policies, e.g., [16, 19]. For analysis and enforcement of security and trust management policies, a method for generating efficient algorithms and implementations from policies specified using logic rules is highly desired.

This paper describes a systematic method for deriving efficient algorithms and precise time complexities from extended Datalog rules as it is applied to the analysis of trust management policies specified in SPKI/SDSI, a well-known trust management framework designed to facilitate the development of secure and scalable distributed computing systems. SPKI/SDSI [10] is based on public keys and incorporates *Simple Public Key Infrastructure* (SPKI) and *Simple Distributed Security Infrastructure* (SDSI). It provides fine-grained access control using local name spaces and a security policy model.

The SPKI/SDSI framework facilitates granting and delegating authorizations, as well as naming. It uses name certificates to define names in principals' local name spaces as keys or other names, and uses authorization certificates to grant authorizations and to delegate the ability to grant authorizations. A principal is authorized to access a resource by an authorization certificate or by a chain of certificates involving naming and delegation. Designing efficient algorithms for inferring authorizations and answering related queries is essential for enforcing SPKI/SDSI policies.

We express policy analysis problems using extended Datalog, which is Datalog [7, 2] extended with list constructors and external functions. We represent certificates as facts, and describe rules and queries for computing the reduction closure, inferring authorizations, and solving other policy analysis problems for SPKI/SDSI. These other analysis problems include ones about the current state of the policy, as well as ones about changes in the state that would be caused by possible changes in the policy, such as expiration or addition of a set of certificates.

We describe our method for systematically generating specialized algorithms and data structures, together with precise time complexity formulas, from extended Datalog rules as it is applied to computing reduction closure and inferring all authorizations. The generated algorithms employ an incremental approach that considers one certificate or intermediate analysis fact at a time, and use a combination of linked and indexed data structures to represent different certificates and intermediate values. The running time is optimal for the respective rules, in the sense that each combination of instantiations of hypotheses is considered once in $O(1)$ time.

We then describe other policy analysis problems as additional rules and queries, and use a method to systematically push given inputs for the analyses from queries into hypotheses of rules, yielding specialized and simplified rules for the given queries. This is similar to pushing demands by queries in magic set transformations [4], but instead of yielding more complicated rules with magic predicates, we obtain simplified, specialized rules that are much easier for generating efficient implementations and precise complexities.

[*] This work was supported in part by NSF under grants CCR-0306399 and CCR-0311512 and ONR under grant N00014-04-1-0722.

[†] Corresponding author.

Contrasting various previous works, our rules and algorithms for policy analysis support all aspects defined in the specification for SPKI/SDSI, including any number of resources and accesses, names consisting of any number of identifiers, and validity intervals. We also have a prototype implementation, and experimental results confirm our precise complexity analysis.

A significant amount of work has been done on algorithms for SPKI/SDSI policy enforcement and analysis [27, 1, 10, 18, 8, 21, 12, 3, 13, 20, 17, 9]. Our approach of expressing policy analysis problems as extended Datalog rules is much simpler than previous techniques for analysis of SPKI/SDSI policies. Our method also derives better, more precise time complexities than before in addition to generating complete algorithms and data structures. The method is general, with many applications beyond policy analysis. It extends our previous method for Datalog [23] to handle list constructors, external functions, and queries.

2. SPKI/SDSI

In SPKI/SDSI systems, *principals* are the users and are identified by public keys, which we will simply refer to as *keys*. *Identifiers* are words over a standard alphabet and are used to refer to principals and resources. A *name* is a key followed by a sequence of identifiers.

SPKI/SDSI certificates are *name certificates* and *authorization certificates*. A *name certificate* defines a local name in its issuer's local name space. A name certificate is the 4-tuple (K, I, S, V), where K is the public key of the issuer of the certificate; I is an identifier from the local name space of the issuer; S is the name or key that the local name KI stands for; V is the validity time interval for the certificate and is of the form $[t1, t2]$, where $t1$ and $t2$ are absolute time constants. The 4-tuple defines the name KI to stand for S during validity interval V. A name certificate can only be issued by the principal to whom the name being defined is local. We refer to certificates in which S is a name as *name-name* certificates, and to ones in which S is a key as *name-key* certificates. A name can correspond to a set of keys.

Principals use *authorization certificates* to grant permissions for accessing resources to other principals. An authorization certificate is a 5-tuple (K, S, D, P, V), where K is the public key of the certificate issuer — the principal granting authorization; S is the subject of the certificate — the key or name that is being granted authorization; D is a boolean delegation bit indicating if the subject is granted the right to delegate the permissions granted by the certificate to others; P is the set of permissions, i.e. operation-resource pairs, being granted; V is a validity interval as for name certificates.

A principal Pr has permission for an operation on a resource if there is a valid authorization certificate (R, Pr, D, P, V), where P contains the operation-resource pair, and R is the owner of the resource involved in permission P, or if such a certificate can be inferred, i.e. there is a chain of certificates that authorizes the access. Certificates are composed in chains by use of the following composition rules.

- Two name certificates, such as $(k1, id1, k2\ id2\ ids, v1)$ and $(k2, id2, s, v2)$, can be composed to infer $(k1, id1, s\ ids, v3)$, where $v3$ is the intersection of validity intervals $v1$ and $v2$.
- Two authorization certificates, $(k1, k2, d1, p1, v1)$, where $d1 = TRUE$, and $(k2, s, d2, p2, v2)$ can be composed to infer the certificate $(k1, s, d2, p3, v3)$, where $p3$ is the intersection of authorization sets $p1$ and $p2$, $v3$ is the intersection of validity intervals $v1$ and $v2$.
- An authorization certificate $(k1, k2\ id\ ids, d, p, v1)$ and a name certificate $(k2, id, s, v2)$, can be composed to infer $(k1, s\ ids, d, p, v3)$, where $v3$ is the intersection of validity intervals $v1$ and $v2$.

The *closure* of a set of certificates contains all given certificates and all certificates that can be inferred using the above rules. However, the closure of a set of certificates may be infinite. The *reduction closure* of a set of authorization and name certificates contains all given certificates and all certificates that can be inferred using chains in which every certificate after the first one has a key as its subject. For each name occurring in a set of certificates, the reduction closure contains all name-key certificates that define the name as a key, as in the full closure of the set of certificates. Also, for a given key, the reduction closure contains all authorization certificates in which the key is a subject, that occur in the full closure. Thus, the reduction closure can be used to find all keys that a name stands for, as well as to find all permissions that a key has.

3. Computing Reduction Closure Efficiently

This section expresses reduction closure and authorization inference using extended Datalog rules, and describes the generation of specialized algorithms and data structures from the rules, we also analyze precisely the time complexities, expressing the complexities in terms of characterizations of the given set of certificates.

3.1 Expressing reduction closure in rules

A Datalog program is a finite set of relational rules of the form

$$p_1(x_{11}, ..., x_{1a_1}), ..., \ p_h(x_{h1}, ..., x_{ha_h}) \rightarrow q(x_1, ..., x_a)$$

where h is a natural number, each p_i (respectively q) is a relation of a_i (respectively a) arguments, each x_{ij} and x_k is either a constant or a variable, and variables in x_k's must be a subset of the variables in x_{ij}'s. If $h = 0$, then there are no p_i's or x_{ij}'s, and x_k's must be constants, in which case $q(x_1, ..., x_a)$ is called a *fact*. For the rest of the paper, "rule" refers only to the case where $h \geq 1$, in which case each $p_i(x_{i1}, ..., x_{ia_i})$ is called a *hypothesis* of the rule, and $q(x_1, ..., x_a)$ is called the *conclusion* of the rule. The meaning of a set of rules and a set of facts is the smallest set of facts that contains all the given facts and all the facts that can be inferred, directly or indirectly, using the rules.

We use the following relations to denote certificates:

- `nameCert(k,id,s,v)`: a given name certificate.
- `authCert(k,s,d,p,v)`: a given authorization certificate.
- `name(k,id,s,v)`: an inferred name certificate.
- `auth(k,s,d,p,v)`: an inferred authorization certificate.

We use three external functions. The symbol | separates the head from the tail in a sequence of identifiers and `NIL` denotes the empty list. The functions `PInt(p1,p2)` and `VInt(v1,v2)` return the intersections of two sets of permissions, and two validity intervals, respectively.

The rules for composing chains of certificates can readily be written as the extended Datalog rules shown in Figure 1.

3.2 Generating efficient algorithms and data structures

We transform the extended Datalog rules into an efficient implementation using the method in [23] for Datalog rules. A small extension is needed to handle the external functions `|`, `VInt`, and `PInt`.

The method has three steps. `Step 1:` transform the least fixed point (LFP) semantics of the extended Datalog rules into a `while`-loop. `Step 2:` transform expensive set operations in the loop into incremental operations. `Step 3:` design appropriate data structures for each set, so that operations on it can be implemented efficiently. These three steps correspond to dominated convergence

```
1. nameCert(k,id,s,v)→name(k,id,s,v).
2. authCert(k,s,d,p,v)→auth(k,s,d,p,v).
3. name(k1,id1,k2|(id2|ids),v1),name(k2,id2,k3|NIL,v2)→name(k1,id1,k3|ids,VInt(v1,v2)).
4. auth(k1,k2|NIL,TRUE,p1,v1),auth(k2,k3|NIL,d2,p2,v2)→auth(k1,k3|NIL,d2,PInt(p1,p2),VInt(v1,v2)).
5. auth(k1,k2|(id|ids),d,p,v1),name(k2,id,k3|NIL,v2)→auth(k1,k3|ids,d,p,VInt(v1,v2)).
```

Figure 1: Rules for computing the reduction closure.

[6], finite differencing [25], and real-time simulation [24], respectively, as studied by Paige et al.

Fixed-point specification and* `while`*-loop We represent a relation of the form `Q(a1, a2, ..., an)` using tuples of the form `[Q a1 a2 ... an]`. S `with` X and S `less` X denote $S \cup \{X\}$ and $S - \{X\}$, respectively. We use the notation $\{X : Y_1 \texttt{ in } S_1, \ldots, Y_n \texttt{ in } S_n | Z\}$ for set comprehension. Each Y_i enumerates elements of S_i; for each combination of values $Y_1, \ldots, Y_n$, if the value of boolean expression Z is true, then the value of expression X forms an element of the resulting set. If $|Z$ is omitted, Z is implicitly the constant `true`.

$\{[X_1\ Y_1] \ldots [X_n\ Y_n]\}$ denotes a map that maps X_1 to Y_1, …, X_n to Y_n. $dom(E)$ denotes the domain set of map E, i.e. $\{X : [X\ Y] \texttt{ in } E\}$. $M\{X\}$ denotes the *image set* of X under map M, i.e. $\{Y : [X\ Y] \texttt{ in } M\}$. $M\{X\} := S$ denotes setting the image set $M\{X\}$, of X under map M, to S. $\texttt{LFP}(S_0, F)$ denotes the smallest set S that satisfies the conditions $S_0 \subseteq S$ and $F(S) = S$.

The algorithm is expressed using standard control constructs `while`, `for`, `if`, and `case`. Program block structure is indicated by indentation. We abbreviate $X := X$ `op` Y as X `op:=` Y.

The input to the algorithm is the given set of certificates represented by a set `certs` of facts. We define `rcerts` to be the set of facts in `certs` represented as tuples as described above.

```
rcerts = {[authCert k s d p v]:
            authCert(k,s,d,p,v) in certs}
         ∪ {[nameCert k id s v] :
            nameCert(k,id,s,v) in certs}.
```

Given any set `R` of facts, and an extended Datalog rule with rule number `n` and with relation `e` in the conclusion, let `ne(R)` be the set of all facts that can be inferred by that rule given the facts in `R`. For our rules we have:

```
1name={[name k id s v] :
   [nameCert k id s v] in R},
2auth={[auth k s d p v] :
   [authCert k s d p v] in R},
3name={[name k1 id1 k3|ids VInt(v1,v2)] :
   [name k1 id1 k2|(id2|ids) v1] in R and
   [name k2 id2 k3|NIL v2] in R},
4auth={[auth k1 k3|NIL d2 PInt(p1,p2) VInt(v1,v2)] :
   [auth k1 k2|NIL TRUE p1 v1] in R and
   [auth k2 k3|NIL d2 p2 v2] in R},
5auth={[auth k1 k3|ids d p VInt(v1,v2)] :
   [auth k1 k2|(id|ids) d p v1] in R and
   [name k2 id k3|NIL v2] in R}.
```

The meaning of the given set of certificates and the extended Datalog rules for reduction closure is:

```
LFP({},F), where F(R)=R∪rcerts ∪
  1name(R)∪2auth(R)∪3name(R)∪4auth(R)∪5auth(R)
                                                  (1)
```

This least-fixed point specification of computing the reduction closure is transformed into the following `while` loop:

```
R := {};
while exists x in F(R) - R:                (2)
  R with := x;
```

The idea behind this transformation is to perform small update operations in each iteration of the `while`-loop. After the execution of this loop `R` contains all facts that are given or can be inferred by the rules. `R` is referred to as the *resultset*.

Incremental computation Next we transform expensive set operations in the loop into incremental operations. The idea is to replace each expensive expression exp in the loop with a variable, say E, and maintain the invariant $E = exp$, by inserting appropriate initializations and updates to E where variables in exp are initialized and updated, respectively.

The expensive expressions in type inference are all sets of facts inferred by each rule and a workset `W`. We use fresh variables to hold each of their respective values and maintain an invariant for each of these sets, in addition to one for the workset.

```
I1name = 1name(R), I2auth = 2auth(R),
I3name = 3name(R), I4auth = 4auth(R),
I5auth = 5auth(R), W = F(R) - R.
```

As an example of incremental maintenance of the value of an expensive expression, consider maintaining the invariant `I3name`. `I3name` is the value of the set formed by joining two name certificates. `I3name` can be initialized to {} with the initialization `R := {}`. To update `I3name` incrementally with update `R with:= x`, if `x` is of the form `[name k1 id1 k2|(id2|ids) v1]`, we consider matching tuples of the form `[name k2 id2 k3|NIL v2]` and add all corresponding new tuples `[name k1 id1 k3|ids VInt(v1,v2)]` to `I3name`. To form the tuples to be added, we need to efficiently find the appropriate values of variables that occur in `[name k2 id2 k3|NIL v2]` tuples, but not in `[name k1 id1 k2|(id2|ids) v1]`, i.e., the values of `k3` and `v2`, so we maintain an auxiliary map, `I3name1`, shown below, that maps `[k2 id2]` to `[k3 v2]`. Symmetrically, if `x` is a tuple of the form `[name k2 id2 k3|NIL v2]`, we need to consider every matching tuple of the form `[name k1 id1 k2|(id2|ids) v1]` and add the corresponding tuple of the form `[name k1 id1 k3|ids VInt(v1,v2)]` to `I3name`, so we maintain the auxiliary map `I3name2` below.

```
I3name1 = {[[k2 id2] [k3 v2]] :
      [name k2 id2 k3|NIL v2] in R},
I3name2 = {[[k2 id2] [k1 id1 ids v1]] :
      [name k1 id1 k2|(id2|ids) v1] in R}.
```

The first set of components in an auxiliary map is referred to as the *anchor* and the second set of elements as the *nonanchor*.

Thus, the algorithm can directly find only matching tuples and consider only combinations of facts that make both hypotheses true simultaneously, and it considers each combination only once. Auxiliary maps are maintained similarly for all maintained invariants, `I4auth` and `I5auth` here, that are formed by joining two relations.

All variables holding the values of expensive computations listed above, and auxiliary maps, are initialized together with the assignment `R := {}` and updated incrementally together with the assignment `R with:= x` in each iteration. We show the update for the addition of a fact of relation `name` only for `I3name` and auxiliary map `I3name2`. Other updates are processed in the same way.

```
case of x of [name k1 id1 k2|(id2|ids) v1]:
 I3name ∪:= {[name k1 id1 k3|ids VInt(v1,v2)]
       : [k3 v2] in I3name1{[k2 id2]}};
 W ∪:= {[name k1 id1 k3|ids VInt(v1,v2)]
       : [k3 v2] in I3name1{[k2 id2]}
       | [name k1 id1 k3|ids VInt(v1,v2)] notin R};
 I3name2 ∪:= {[[k2 id2] [k1 id1 ids v1]]};
                                                    (3)
```

Adding these initializations and updates, and other similar ones for the other cases, and replacing `F(R) - R` with `W` in (2), we obtain the following complete code:

```
initialization; R:={};
while exists x in W:
   update using (3) and                        (4)
   similar updates for the other cases;
   W less:= x; R with:= x;
```

Next, we eliminate dead code. To compute the resultset `R`, only `W` and the auxiliary maps are needed; the invariants maintained, i.e., `I3name`, `I4auth`, and `I5auth`, are dead because `F(R)-R` in the `while` loop was replaced with `W`. We eliminate them from the initialization and updates. For example, eliminating them from the updates in (3), we eliminate lines 2-3.

```
case of x of [name k1 id1 k2|(id2|ids) v1]:
 W ∪:= {[name k1 id1 k3|ids VInt(v1,v2)]
       : [k3 v2] in I3name1{[k2 id2]}
       | [name k1 id1 k3|ids VInt(v1,v2)] notin R};
 I3name2 ∪:= {[[k2 id2] [k1 id1 ids v1]]};
                                                    (5)
```

We clean up the code to contain only uniform operations on set elements. This simplifies data structure design. We decompose `R` and `W` into several sets, each corresponding to one relation in the extended Datalog rules. `R` is decomposed to `RnameCert`, `RauthCert`, `Rname` and `Rauth`; `W` is decomposed to `WnameCert`, `WauthCert`, `Wname`, and `Wauth`. This decomposition lets us eliminate relation names from the first component of tuples, with appropriate changes to the rest of the code. Then, we apply the following three sets of transformations.

`(i)` Transform operations on sets into loops that use operations on set elements. Each addition of a set is transformed to a `for`-loop that adds the elements one at a time. For example, lines 2-4 of (5) are transformed into:

```
 for [k3 v2] in I3name1{[k2 id2]}:
   if [k1 id1 k3|ids VInt(v1,v2)] notin Rname:
     Wname with:= [k1 id1 k3|ids VInt(v1,v2)];
                                                    (6)
```

`(ii)` Replace tuples and tuple operations with maps and map operations. Specifically, replace all `for`-loops as follows. (6) is transformed into:

```
for [k2 id2] in dom(I3name1):
 for [k3 v2] in I3name{[k2 id2]}:
  if [k1 id1 k3|ids VInt(v1,v2)] notin Rname:
   Wname with:= [k1 id1 k3|ids VInt(v1,v2)];
```

We replace the `while` loop similarly. Also, we replace each $[X\ Y]$ `notin` M with Y `notin` $M\{X\}$. Each addition to a map M `with:=` $[X\ Y]$ is replaced with $M\{X\}$ `with:=`Y.

`(iii)` Test for membership before adding or deleting an element to or from a set. Specifically, we replace each statement S `with:=`X with `if` X `notin` S `:` S `with:=`X.

Note that when removing an element from a workset, the membership test is unnecessary, since the element is retrieved from the workset. Also, when adding an element to a resultset, the membership test is unnecessary, since elements are moved from the corresponding workset to the resultset one at a time, and each element is put in the workset and thus in the resultset only once.

After the above transformations, each firing of an extended Datalog rule involves a constant number of set operations. Since each set operation takes worst-case constant time in the generated code, as described below, each firing takes worst-case constant time. The complete pseudocode for computing reduction closure efficiently is shown in figure 2.

Data structures We describe how to guarantee that each set operation takes worst-case constant time. The operations are of the following kinds: set initialization $S := \{\}$, computing image set $M\{X\}$, element retrieval `for` X `in` S and `while exists` X `in` S, membership test X `in` S and X `notin` S, element addition S `with` X, and element deletion S `less` X. Membership test and computing image set are called *associative access*.

A uniform method is used to represent all sets and maps, using arrays for sets that have associative access, linked lists for sets that are traversed by loops, and both arrays and linked lists for sets that have both operations.

Resultsets are represented by nested array structures. A resultset containing tuples with a components is represented using an a-level nested array structure. The first level is an array indexed by values in the domain of the first component of the resultset; the k-th element of the array is null if there is no tuple in the resultset whose first component has value k, and otherwise is `true` if $a = 1$, and otherwise is recursively an $(a-1)$-level nested array structure for the remaining components of tuples in the resultset whose first component has value k.

Worksets corresponding to relations that occur in the conclusions of rules are represented by arrays and linked lists. Each workset is represented the same way as the corresponding resultset with two additions. First, for each array we add a linked list containing indices of non-null elements of the array. Second, to each linked list we add a tail pointer, i.e., a pointer to the last element, so the list can be used as a queue. One or more records are used to put each array, linked list, and tail pointer together. Each workset corresponding to a relation that does not occur in the conclusion of any rule, is represented simply as a nested queue structure (without the underlying arrays), one level for each component of the tuples, linking the elements (instead of array indices) directly.

Auxiliary maps are implemented as follows. Each auxiliary map for a relation that appears in an extended Datalog rule's conclusion uses a nested array structure for all components of the tuples and additionally linked lists for each non-anchor component. Each auxiliary map for a relation that does not appear in the conclusion of any rule uses a nested array structure for the anchor components, and nested linked-lists for the non-anchor components.

3.3 Time complexity analysis

We analyze the time complexity of computing reduction closure by carefully bounding the number of facts actually used by the rules. For each rule we determine precisely the number of facts processed by it, avoiding where possible approximations that use the product of the sizes of individual argument domains.

```
W := rcerts;
I3name1 := {}; I3name2 := {};
I4auth1 := {}; I4auth2 := {}; I5auth3 := {};
R := {};

while exists x in W:

  case x of [nameCert k id s v]:
    if [name k id s v] notin R:
    W with := [name k id s v];

  case x of [authCert k s d p v]:
    if [auth k s d p v] notin R:
      W with := [auth k s d p v];

  case x of [name k1 id1 k2|(id2|ids) v1]:
    W U:= {[name k1 id1 k3|ids VInt(v1,v2)] : [k3 v2] in I3name1{[k2 id2]}
                                         | [name k1 id1 k3|ids VInt(v1,v2)] notin R};
    I3name2 with:= [[k2 id2] [k1 id1 ids v1]];

  case x of [name k2 id2 k3|NIL v2]:
    W U:= {[name k1 id1 k3|ids VInt(v1,v2)] : [k1 id1 ids v1] in I3name2{[k2 id2]}
                                          | [name k1 id1 k3|ids VInt(v1,v2)] notin R};
    W U:= {[auth k1 k3|ids d p VInt(v1,v2)] : [k1 ids d p v1] in I5auth3{[k2 id2]}
                                          | [auth k1 k3|ids d p VInt(v1,v2)] notin R};
    I3name1 with:= [[k2 id2] [k3 v2]];

  case x of [auth k1 k2|NIL TRUE p1 v1]:
    W U:= {[auth k1 k3|NIL d2 PInt(p1,p2) VInt(v1,v2)] : [k3 d2 p2 v2] in I4auth2{[k2]}
                                                  | [auth k1 k3|NIL d2 PInt(p1,p2) VInt(v1,v2)] notin R};
    I4auth1 with:= [[k2] [k1 TRUE p1 v1]];

  case x of [auth k2 k3|NIL d2 p2 v2]:
    W U:= {[auth k1 k3|NIL d2 PInt(p1,p2) VInt(v1,v2)] : [k1 d1 p1 v1] in I4auth1{[k2]}
                                                  | [auth k1 k3|NIL d2 PInt(p1,p2) VInt(v1,v2)] notin R};
    I4auth2 with:= [[k2] [k3 d2 p2 v2]];

  case x of [auth k1 k2|(id|ids) d p v1]:
    W U:= {[auth k1 k3|ids d p VInt(v1,v2)] : [k3 v2] in I3name1{[k2 id]}
                                          | [auth k1 k3|ids d p VInt(v1,v2)] notin R};
    I5auth3 with:= [[k2 id] [k1 ids d p v1]];

  W less:= x;
  R with:= x;
```

Figure 2: Pseudocode for computing reduction closure.

We first define the size parameters used in the complexity analysis. The number of facts of a relation r that are given or can be inferred is called r's *size*. For a relation named r, #r denotes the size of r. We use the following size parameters about inferred certificates:

- nameKey — number of name certificates that have keys as subjects.
- nameKeyPerName — maximum number of name certificates, that have keys as subject, for one name.
- namePerSubject — maximum number of name certificates for one subject.
- authD — number of authorization certificates with a delegation bit TRUE.
- authPerIssuer — maximum number of authorization certificates for one issuer.
- authPerIssuerD — maximum number of authorization certificates with delegation bit TRUE for one issuer.
- authPerSubject — maximum number of authorization certificates for one subject.

In addition, we use key for the total number of different keys in the given certificates.

The time complexity for a set of Datalog rules is the total number of combinations of hypotheses considered in evaluating the rules. For each rule r, the number of firings for the rule is: (i) for rules with one hypothesis: the number of facts which make the hypothesis true; (ii) for rules with two hypotheses: the number of combinations of facts that make the two hypotheses simultaneously true. The total time complexity is time for reading the input, plus the time for firing all rules.

The total time complexity for computing the reduction closure is time for reading the input, which is $O($`#authCert` $+$ `#nameCert`$)$, plus the time for applying each of the rules. `VInt(v1, v2)` is computed in constant time; `PInt(p1,p2)` can be computed in time $O($`p`$)$, where `p` is maximum size of a permission argument in the given authorization certificates. List operations involving `|` can be performed in time $O(1)$.

Time complexity of the rules used for name-reduction closure and inferring authorizations is as follows:

1. $O($`#nameCert`$)$
2. $O($`#authCert`$)$
3. $O(\min($`#name` $\times$ `nameKeyPerName`, `nameKey` $\times$ `namePerSubject`$))$
4. $O(\min($`authD` $\times$ `authPerIssuer`, `#auth` $\times$ `authPerIssuerD`$))$
5. $O(\min($`#auth` $\times$ `nameKeyPerName`, `nameKey` $\times$ `authPerSubject`$))$

The time complexity for the whole reduction closure is the sum of the time complexities for rules 1, 2, 3, 4, and 5. The sum for rules 1 and 2 is the number of given certificates. The sum for rules 3-5 is larger and decides the total time complexity.

To compare with previous results, suppose we eliminate the permission and validity interval arguments, or consider only certificates with the given permission and validity interval, as in [8, 17]. Then `nameKeyPerName` is the maximum number of keys a single local name reduces to, and is `key` in the worst case; and `authPerIssuerD` is the maximum number of keys authorized by one issuer with delegation bit TRUE, and is again `key` in the worst case. Thus, our precise complexity formulas for rules 3-5 is $O(($`#name` $+$ `#auth`$)$ $\times$ `key`$)$ in the worst case. `#name`$+$`#auth` is the total number of certificates inferred and, as noted in [17], is bounded by `in`$\times$`key`, where `in` is the size of the input, i.e., the sum of the sizes of the given certificates; note that the size of a certificate might not be a constant because its subject may be a key followed by a list of identifiers. Therefore, the time complexity $O($`in` $\times$ `key`$^2)$ from previous work [17] is an upper bound of our more precise complexity analysis.

4. Specialized Policy Analysis Problems

This section discusses how to solve specialized certificate analysis problems and analyze their algorithm complexities. The algorithms for computing reduction closure can be used to solve specialized analysis problems. However, these algorithms compute all authorizations and all name-key correspondences, given a set of certificates. This may be unnecessary, since many policy analysis problems require computing only a few authorizations or resolving only a few names. Therefore, we use specialized extended Datalog rules for the specialized analysis problems; these specialized rules can be used to generate an efficient algorithm for each analysis problem, and infer only the authorizations and resolve only the names needed for that problem. Also, the original reduction closure algorithm does not give a direct way of solving some important policy analysis problems, specifically when questions about name certificates are asked, when sets of resources or keys are given. There are algorithms for solving these problems in [17], but these require complex pushdown system structures that are not inherent to the problems' structure.

We first introduce extended Datalog rules to solve the problems and then show a way to construct specialized rules from given rules, by pushing the constants bound by the query into the rules. There are automatic ways of generating on-demand rules such as *Magic Set Transformation* (MST) [4, 26]. MST introduces demand relations corresponding to the query; and makes changes that limit the facts being inferred to ones demanded by the query. We chose to push the constants in a naïve manner despite the fact that MST may do better than our technique for some problems; mainly because MST is much more sophisticated, the order of hypotheses in the original rules may significantly change the efficiency of the transformed rules and moreover there is no reason (except the resulting complexity) to prefer an order to the other before starting the transformation. By pushing constants into the rules, we obtain simpler rules and precise complexities.

4.1 Policy analysis problems and complexity analysis in a logic framework

We consider all the analysis problems studied in [17]. All problems are solved with respect to a given set of certificates. In the rules, we use the names "permissions" and "resources" interchangably, in our context "permission" means an access to a resource without loss of generalization. In the construction of rules, we leave the unknown to the question as the last argument, and try to remain consistent on the order of keys, permissions, etc. otherwise. The relations are named as close to the real meaning of the relation, e.g. `canAccess(K,P)` stands for the relation "a key K is authorized for permission P". For each problem, we first give a set of rules, followed by a Prolog-like query, that will return the requested result.

Figure 3 shows all of the rules for the problems below. In the rules, `owner(o,p)` denotes that `o` is an owner of permission `p`, and `auth` is as defined before. In the last four analysis problems, where some certificates are removed, `canAccess2` is defined in a similar way as `canAccess` in the first analysis, but uses authorizations inferred using only the remaining certificates, i.e., using an `auth2` relation computed as the reduction closure of the remaining certificates.

We introduce the notation for the auxiliary values used for the complexity analysis.

- `authPerKey` is the maximum number of authorizations that has a specific key as a subject.
- `ownersPerRes` is the maximum number of owners for a single resource.
- `l` is the maximum number of identifiers occurring in a name that is the subject of a certificate.
- `identifiers` is the number of distinct identifiers occurring in the certificates.
- `len(N)` for a name N is the length of the name N.

Authorized Access 1: Is a principal K authorized to permission P? This is determined in time $O($`ownersPerRes` $\times$ `authPerKey`$)$.
Authorized Access 2: Given a permission P and name N, is N authorized to P?
Authorized Access 3: Given a permission P, what names are authorized to access P?
These two questions are answered the same way as question 1; the preprocessing for adding the certificates for reduction closure takes linear time in the length of the name N for question 2; and for question 3 this procedure needs to assign a valid string of identifiers of at most length `l`, which would take `key` $\times$ `identifiers`$^{\texttt{l}}$, but the key is not bounded either so instead of `authPerKey` as a factor, we have `#auth`. Notice that this exponential behaviour for the third question comes from the nature of the problem, since the set of names authorized to access P might be an infinite set. So the precise complexities for question 2 and 3 respec-

tively are: $O(\texttt{ownersPerRes} \times \texttt{\#authPerKey} + \texttt{len(N)})$ and $O(\texttt{ownersPerRes} \times \texttt{\#auth} + \texttt{key} \times \texttt{identifiers}^{\texttt{l}})$.

Shared Access 1: Given two permissions P_1 and P_2, which principals are authorized for both? A straightforward analysis just as above shows that the complexity for the solution to this problem is $O(\texttt{key} + \texttt{\#auth} \times \texttt{ownersPerRes})$.

Shared Access 2: Given two principals K_1 and K_2 and a permission P, is both K_1 and K_2 authorized for P? This question is answered in a constant time factor of the answer to the Authorized Access 1 question, so it takes time $O(\texttt{ownersPerRes} \times \texttt{authPerKey})$.

Shared Access 3: Given two principals K_1 and K_2 and a finite set of permissions $Ps = \{P_1, ..., P_n\}$, what is the subset of Ps that K_1 and K_2 are authorized for? This question is answered using the rule in Shared Access 2, by checking for all elements in Ps, but the permission is not bound for `canAccess`, so it takes $O(\texttt{n} + \texttt{\#owner} \times \texttt{authPerKey})$ time.

Compromisation Assessment 1 (also called Expiration Vulnerability 1): What permissions from a finite set of permissions $Ps = \{P_1, ..., P_n\}$ would a given principal K lose authorization for, if a subset C' of the original certificate set C were to be removed? This question is answered using `canAccess` without `p` being bound, checked for each element in Ps, so it takes $O(\texttt{n} + \texttt{\#owner} \times \texttt{authPerKey})$ time (since `#auth2`, the number of authorizations inferred not using C' is less than `#auth`, we can ignore that part).

Compromisation Assessment 2 (also called Expiration Vulnerability 2): What principals would have lost authorization for a permission P if a subset C' of the original certificate set C were to be removed? This question is answered using the rule in Authorization Access 1 without binding `k`, by checking for all keys in the system, so it is answered in $O(\texttt{key} + \texttt{ownersPerRes} \times \texttt{\#auth})$ time (since `#auth2`, the number of authorizations inferred not using C' is less than `#auth`, we can ignore that part).

Universally Guarded Access 1: Must all authorizations for permission P involve a certificate signed by principal K? We answer the negation of this question for simplicity, in other words our rule gives a "no" for a "yes" instance and vice versa. This question is answered using the rule in Authorization Access 1 without binding `k`, by checking for all keys in the system, so it is answered in $O(\texttt{key} + \texttt{ownersPerRes} \times \texttt{\#auth})$ time (since `#auth2`, the number of authorizations inferred not using certificates signed by K is less than `#auth`, we can ignore that part).

Universally Guarded Access 2: Must all authorizations that grant a given principal K' a finite set of permissions $Ps = \{P_1, ..., P_n\}$ involve a certificate signed by K? Again we answer the negation of this question for simplicity. This question is answered using the rule in Authorization Access 1 without binding `p`, by checking for all elements in Ps, so it takes $O(\texttt{n} + \texttt{\#owner} \times \texttt{authPerKey})$ time (since `#auth2`, the number of authorizations inferred not using certificates signed by K is less than `#auth`, we can ignore that part).

4.2 Constructing specialized rules

We demonstrate how to push constants to create specialized rules on one of the analysis problems. Consider the rule set and the query for the problem Compromisation Assessment 2:

$$\begin{array}{l}\texttt{canAccess(k,p,t)}, \neg\texttt{canAccess2(k,p,t)} \\ \quad \rightarrow\texttt{compromisedPrinciples(p,t,k)} \\ \text{Query}: \texttt{compromisedPrinciples(P,T,k)}.\end{array}$$

Now since the permission P and time T is given when the question is asked, we push them inside the relations on the right hand side, yielding:

$$\begin{array}{l}\texttt{canAccess(k,P,T)}, \neg\texttt{canAccess2(k,P,T)} \\ \quad \rightarrow\texttt{compromisedPrinciples(P,T,k)}\end{array}$$

Now it is easy to observe that the conclusion expresses the constants unnecessarily, since the hypotheses are already aware of the values of them. So we can rewrite :

$$\begin{array}{l}\texttt{canAccess(k,P,T)}, \neg\texttt{canAccess2(k,P,T)} \\ \quad \rightarrow\texttt{compromisedPrinciples\_PT(k)}\end{array}$$

This new rule is the `compromisedPrinciples` rule specialized to constants P and T; it returns precisely what we are looking for, the resulting keys. Notice that this push-and-specialize method can be applied iteratively in general, and it is particularly simple in this case since there is no recursion. In other words, in this example `canAccess(k,P,T)` can be rewritten as `canAccess_PT(k)` by pushing the constants into hypotheses properly.

5. Experimental Results

To experimentally confirm our time complexity calculations, we generated an implementation of our algorithm for computing reduction closure in Python. The generated implementation consists of 180 lines of Python code. We analyzed sets of certificates of varying sizes, to determine how the running times of the algorithms scale with the number of given certificates. For each certificate set, we report the CPU time for the analysis, using Python 2.3.5 on a 1.73 GHz Pentium M processor, with 366 MHz 448 MB RAM, running Windows XP. Reported times are averaged over 10 trials.

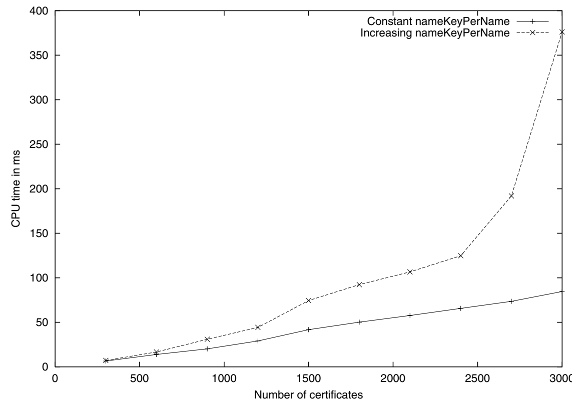

Figure 4: Time to infer all name certificates only.

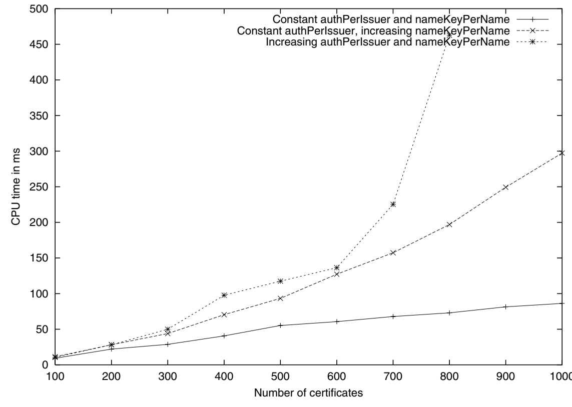

Figure 5: Time to infer authorization certificates after name certificates have been inferred.

For the experiments we first infer all name facts using rules 1 and 3, and then infer the authorizations by rules 2,4 and 5. This

Authorized Access 1:
`owner(o,p), auth(o,k|NIL,d,ps,v), p in ps, t in v` $\rightarrow$ `canAccess(k,p,t).`
Query: `canAccess(K,P,T).`

Authorized Access 2:
Suppose the asked given name is $\mathtt{N} = \mathtt{K}\ \mathtt{I}_1\ \mathtt{I}_2\ ..\ \mathtt{I_n}$, add new name certificates:
$(\mathtt{K}, \mathtt{I}_1, \mathtt{K}_1|\mathtt{NIL}, \mathtt{V}), (\mathtt{K}_1, \mathtt{I}_2, \mathtt{K}_2|\mathtt{NIL}, \mathtt{V}), ...(\mathtt{K_{n-1}}, \mathtt{I_n}, \mathtt{K_n}|\mathtt{NIL}, \mathtt{V})$, where each $\mathtt{K}_i$ is a fresh key, `V` is a validity interval satisfied at the current time `T`.
Query: `canAccess(`$\mathtt{K_n}$`,P,T).`

Authorized Access 3:
For all keys and identifiers, construct all possible names up to `l` identifiers, and add new name certificates corresponding to these names as shown above.
Query: `canAccess(k,P,T).`

Shared Access 1:
`canAccess(k,p1,t), canAccess(k,p2,t)` $\rightarrow$ `sharingPrinciple(p1,p2,t,k).`
Query : `sharingPrinciple(P1,P2,T,k).`

Shared Access 2:
`canAccess(k1,p,t), canAccess(k2,p,t)` $\rightarrow$ `sharingResource(k1,k2,p,t).`
Query : `sharingResource(K1,K2,P,T).`

Shared Access 3:
`p in ps, sharingResource(k1,k2,p,t)` $\rightarrow$ `sharingResources(k1,k2,ps,t,p).`
Query : `sharingResources(K1,K2,{P1,P2,...,Pn},T,p).`

Compromisation Assessment 1:
`p in ps, canAccess(k,p,t),` $\neg$`canAccess2(k,p,t)` $\rightarrow$ `compromisedResource(k,ps,t,p).`
Query : `compromisedResource(K,{P1,P2,..,Pn},T,p).`

Compromisation Assessment 2:
`canAccess(k,p,t),` $\neg$`canAccess2(k,p,t)` $\rightarrow$ `compromisedPrinciple(p,t,k).`
Query : `compromisedPrinciple(P,T,k).`

Universally Guarded Access 1:
`canAccess(k1,p,t), canAccess2(k1,p,t)` $\rightarrow$ `needNotInvolve(p,t).`
Query : `needNotInvolve(P,T).`

Universally Guarded Access 2:
`p in ps, canAccess(k,p,t), canAccess2(k,p,t)` $\rightarrow$ `needNotInvolveMultiple(k,ps,t).`
Query : `needNotInvolveMultiple(K,{P1,P2,..,Pn},T).`

Figure 3: Rules and queries for solving policy analysis problems.

does not affect the resulting facts and was just done for the purpose of having separate experiments for the two parts of the algorithm, so that the effect of changing certain parameters can be seen. Also, the data was generated in such a way that the number of given and inferred certificates are of the same order.

Figure 4 shows the running times for inferring all name certificates. Two series of sets of certificates were used. In both series the number of certificates increases, however in the first one `nameKeyPerName` remains constant. In the second test series `nameKeyPerName` increases as the number of given certificates increase, and both of these parameters increase at the same rate. The results show that CPU time for inferring all name certificates is linear in the number of given name certificates, if `nameKeyPerName` is a constant. Figure 5 shows the running times for inferring all authorization certificates, once name certificates have been inferred. Three series of sets of certificates were used. In all three series the number of given certificates increases, however in the first one `authPerIssuer` and `nameKeyPerName` remain constant. In the second test series only `authPerIssuer` remains constant, while `nameKeyPerName` increases as the number of given certificates increases, and both of these parameters increase at the same rate. In the third series both `authPerIssuer` and `nameKeyPerName` increase along with the number of given certificates and at the same rate as the number of given certificates. The results show that CPU time for inferring all authorization certificates is linear in the number of given certificates, if `authPerIssuer` and `nameKeyPerName` are kept constant; CPU time grows faster if only `authPerIssuer` remains constant. These experimental results confirm our time complexity analysis results. In all experiments the search space increases along with the number of given certificates, so that the test results are not influenced by a disproportionately small search space. The data was generated so that the ratio of the number of keys to the number of given certificates remains the same.

6. Related Work and Conclusion

Surveys of trust management are presented in [11, 5, 19]. Li et al [22] define security analysis problems for trust management systems and analyze their complexity.

SDSI was proposed by Rivest and Lampson [27], as a public-key infrastructure that uses linked local names. SPKI was developed concurrently; it emphasized delegation of authorizations. These two infrastructures were merged to create SPKI/SDSI, described in RFC 2693 [10].

The reduction closure that our algorithm computes includes all the certificates inferred by previous algorithms [8, 17]. Clarke et al. [8] analyze the time complexity of their algorithm to be $O(\mathtt{n}^3 \times \mathtt{l})$, where n is the number of given certificates, and l is the length of the longest subject in any given certificate. Jha and Reps [17] give a more precise bound of $O(\mathtt{in} \times \mathtt{key}^2)$, where in is the size of the input and is bound by $\mathtt{n} \times \mathtt{l}$; it is more precise because key is bound by $O(\mathtt{n})$. Our complexity analysis is even more precise because one of their key factors is nameKeyPerName in our complexity formula, which corresponds to the maximum number of keys a single local name reduces to; while this could be key in the worst case, it is much smaller on average in practice and is close to a constant in large systems. Thus, our complexity analysis is more precise and informative than using only worst-case sizes.

A different algorithm for certificate chain discovery is presented by Li et al [21]. The algorithm as described does not accommodate names containing more than one identifier, but it could be altered to do so. It combines forward and backward search in a graph representation of credentials, and has a time complexity equal to that of the algorithm in [8]. Halpern and Meyden [13, 12] define a semantics for SPKI and SDSI, that facilitates reasoning about SPKI's and SDSI's design. A first-order logic semantics for SPKI/SDSI is presented in [20] and is used to analyze the design of SPKI/SDSI.

Policy analysis for SPKI/SDSI has also been studied. Jha and Reps [17] establish a connection between SPKI/SDSI and pushdown systems, and use existing algorithms for model checking pushdown systems to solve analysis problems for SPKI/SDSI. A similar approach is used in [9], but in addition, propertied of an SPKI/SDSI policy are expressed using a first order temporal logic.

What distinguishes our work is that first we use a novel implementation strategy for reduction closure and inferring authorization that combines an intuitive definition of certificate composition in rules and a systematic method for deriving efficient algorithms and data structures from the rules [23]. The time complexity is calculated directly from the rules, based on a thorough understanding of the algorithms and data structures generated, reflecting the complexities of implementation back into the rules. We also solve known policy analysis problems in a logic framework, and show a straightforward way of constructing specialized rules from our proposed solutions, which allows for easy bottom-up computation of results. Furthermore, we present precise time complexities for our proposed solutions. We achieve more precise worst-case time complexity guarantees that those in previous work. Moreover, our algorithms for authorization and other analysis support any number of resources and types of access, as well as validity intervals and delegation; all aspects defined in the SPKI/SDSI specification.

This method of generating efficient implementations from rules has also been applied to problems beyond the area of policy analysis. In the model checking area, the method is used to derive an efficient algorithm with improved complexity analysis for linear temporal logic model checking of pushdown systems [14]. This model checking framework can express and check many practical properties of programs, including many dataflow properties and general correctness and security properties. For secure information flow analysis, the method was used to develop the first linear-time algorithm for inferring information flow types of programs for a formal type system [15]. The algorithm is also extended with informative error reporting to facilitate error detection and corrections.

References

[1] M. Abadi. On SDSI's linked local name spaces. *Journal of Computer Security*, 6(1-2):3–21, 1998.

[2] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[3] S. Ajmani, D. E. Clarke, C.-H. Moh, and S. Richman. ConChord: Cooperative SDSI certificate storage and name resolution. In *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*, pages 141–154, London, UK, 2002. Springer-Verlag.

[4] C. Beeri and R. Ramakrishnan. On the power of magic. *Journal of Logic Programming*, 10(3-4):255–299, 1991.

[5] P. A. Bonatti and P. Samarati. Logics for authorization and security. In *Logics for Emerging Applications of Databases*, pages 277–323, 2003.

[6] J. Cai and R. Paige. Program derivation by fixed point computation. *Science of Computer Programming*, 11(3):197–261, 1989.

[7] S. Ceri, G. Gottlob, and L. Tanca. *Logic Programming and Databases*. Springer-Verlag New York, Inc., New York, NY, USA, 1990.

[8] D. E. Clarke, J.-E. Elien, C. M. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 9(4):285–322, 2001.

[9] A. K. Eamani and A. P. Sistla. Language based policy analysis in a SPKI trust management system. In *4th Annual PKI Research and Development Workshop: Multiple Paths to Trust, NIST*, pages 162–176, Gaithersburg, MD, 2005.

[10] C. Ellison, B. Frantz, B. Lampson, R. L. Rivest, B. Thomas, and T. Ylonen. RFC 2693: SPKI certificate theory, Sept. 1999.

[11] T. Grandison and M. Sloman. A survey of trust in internet applications. *IEEE Communications Surveys and Tutorials*, 3(4), 2000.

[12] J. Y. Halpern and R. van der Meyden. A logic for SDSI's linked local name spaces. *Journal of Computer Security*, 9(1-2):105–142, 2001.

[13] J. Y. Halpern and R. van der Meyden. A logical reconstruction of SPKI. *Journal of Computer Security*, 11(4):581–614, 2003.

[14] K. Hristova and Y. A. Liu. Improved algorithm complexities for linear temporal logic model checking of push down systems. In *Proceedings of the 7th International Conference on Verification, Model Checking and Abstract Interpretation*, volume 3855, pages 190–206, 2006.

[15] K. Hristova, T. Rothamel, Y. A. Liu, and S. D. Stoller. Efficient type inference for secure information flow. Technical Report DAR 07-35, Computer Science Department, SUNY Stony Brook, May 2007. A preliminary version of this work appeared in *PLAS'06: Proceedings of the 2006 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security*.

[16] S. Jajodia, P. Samarati, and V. S. Subrahmanian. A logical language for expressing authorizations. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 31–42, 1997.

[17] S. Jha and T. W. Reps. Model checking SPKI/SDSI. *Journal of Computer Security*, 12(3-4):317–353, 2004.

[18] N. Li. Local names in SPKI/SDSI. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop (SCFW)*, page 2, Washington, DC, USA, 2000. IEEE Computer Society.

[19] N. Li and J. C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *Proceedings of Practical Aspects of Declarative Languages (PADL)*, pages 58–73, 2003.

[20] N. Li and J. C. Mitchell. Understanding SPKI/SDSI using first-order logic. In *Proceedings of IEEE Computer Security Foundations Workshop (CSFW)*, pages 48–64, 2003.

[21] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed

credential chain discovery in trust management: extended abstract. In *Proceedings of the 8th ACM Conference on Computer and Communications Security (CCS '01)*, pages 156–165, New York, NY, USA, 2001. ACM Press.

[22] N. Li, W. H. Winsborough, and J. C. Mitchell. Beyond proof-of-compliance: Safety and availability analysis in trust management. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 123–139, 2003.

[23] Y. A. Liu and S. D. Stoller. From datalog rules to efficient programs with time and space guarantees. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming*, pages 172–183. ACM Press, 2003.

[24] R. Paige. Real-time simulation of a set machine on a ram. In *Proceedings of the International Conference on Computing and Information*, volume 2, pages 68–73, 1989.

[25] R. Paige and S. Koenig. Finite differencing of computable expressions. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(3):402–454, 1982.

[26] R. Ramakrishnan. Magic templates: A spellbinding approach to logic programs. *Journal of Logic Programming*, 11(3 and 4):189–216, 1991.

[27] R. L. Rivest and B. Lampson. SDSI – A simple distributed security infrastructure. *Presented at the Sixteenth Annual Crypto Conference (CRYPTO '96) Rumpsession, 1996.*