SD3: a trust management system with certified evaluation

Trevor Jim
AT&T Labs Research
180 Park Avenue
Florham Park, NJ 07932

Abstract

We introduce SD3, a trust management system consisting of a high-level policy language, a local policy evaluator, and a certificate retrieval system. A unique feature of SD3 is its certified evaluator: as the evaluator computes the answer to a query, it also computes a proof that the answer follows from the security policy. Before the answer is returned, the proof is passed through a simple checker, and incorrect proofs are reported as errors. The certified evaluator reduces the trusted computing base and greatly increases our confidence that the answers produced by the evaluator follow from the specification, despite complex optimizations.

To illustrate SD3's capabilities, we show how to implement a secure name service, similar to DNSSEC, entirely in SD3.

1 Introduction

This paper introduces SD3, a high-level language for specifying security policies. SD3 is similar to trust management systems like PolicyMaker, SDSI, SPKI, and KeyNote [9, 20, 12, 8]. However, unlike these systems, SD3 has a built-in certificate distribution component, so that it is possible to build a complete security infrastructure within SD3.

To illustrate the capabilities and advantages of SD3, we describe how it can be used to implement a simplified version of a real security infrastructure: DNSSEC, the Domain Name System with security extensions. Our current SD3 prototype captures the core features of DNSSEC, but it uses a different message format, and it lacks features, such as replication and update, that are critical to operating DNS at its present scale (80 million entries as of January 2000). For these reasons, we do not expect to replace BIND, the most popular DNS implementation. Nevertheless, the SD3 approach has many advantages over a system like DNS and BIND:

- SD3 uses a technique called **certified evaluation** to ensure that the answers it produces actually follow from the security policy. This is a critical issue as security infrastructures become larger. For example, in a recent interview about BIND version 9, Conrad says,

  > "The complexity of the recent IETF standards results in a lot of complexity in the code which increases the potential for bugs (security related or otherwise). Standards bloat, in combination with creeping featurism demanded of us by our funders or users, results in software bloat which is probably our most significant challenge." [1]

  Like BIND, our SD3 implementation employs some sophisticated evaluation techniques, including incremental query optimization, cryptographically protected message exchanges, pushed certificates, and local certificate caches. Moreover, we intend to add even more optimizations to make our implementation more efficient and more scalable.

  These improvements will add complexity, but certified evaluation ensures that the answers our implementation produces are nevertheless correct. Our certified evaluator computes not only an answer, but also a *proof* that the answer is correct. Before the answer is returned, the proof must pass through a very simple proof checker. If the proof cannot be checked, the evaluator fails. The correctness of the complete implementation therefore does not depend on the complex evaluator, but rather on the simple checker. This reduces the size of the trusted compute base significantly.

- SD3 is a **high-level** policy language. An SD3 policy abstracts away from details such as certificate distribution and signature verification; these are handled by the underlying policy engine. Consequently, SD3 policies are easy to write and understand. We will demonstrate

[1] LinuxSecurity.com, October 3, 2000.

this by writing the core of a DNSSEC resolver in less than 10 lines of code.

- SD3 is **programmable**. Like other trust management systems, it can easily be retargeted to security policies other than DNSSEC; in fact, it was not developed with DNSSEC in mind. SD3 should be able to support any policy written in SPKI or KeyNote (but not PolicyMaker, which is much more expressive than these other systems). Programmability means that new policies can be implemented more quickly, either from scratch or adapted from existing policies. For example, the DNSSEC resolver we present in this paper was obtained from a DNS resolver by adding a few additional lines of code. In contrast, the real-world transition from DNS to DNSSEC is still not widely implemented, despite years of planning.

The rest of the paper is organized as follows. In Section 2, we introduce SD3. In Section 3 we briefly review DNS before showing how we implement the core of DNSSEC in Section 4. Section 5 gives an informal analysis of the resulting DNSSEC implementation. Section 6 describes the format of our proofs and how the proof checker works. Section 7 describes related work, and we conclude in Section 8.

2 SD3

SD3 is an extension of datalog, a database programming language; it stands for Secure Dynamically Distributed Datalog. We will introduce SD3 in stages, starting with its datalog core, then moving on to our security extensions.

A datalog program is a set of rules. Each rule is a logical implication, written from right to left: if everything on the right holds, then the left-hand side holds as well. For example, the following rules define a graph and its transitive closure:

```
E(1,2) :- ;
E(2,3) :- ;
T(x,y) :- E(x,y);
T(x,y) :- T(x,z), T(z,y);
```

The rules define two relations, `E` and `T`. They say that `E(1,2)` and `E(2,3)` always hold, because their right-hand sides (to the right of the symbol '`:-`') are empty. If we think of `E` as the edge relation of a graph, then the rules define `T` to be its transitive closure: every edge of `E` is an edge of `T` by the third rule, and the fourth rule closes `T` under transitivity. Given this program, datalog can evaluate a query (such as `T(1,x)`) to all of its provable instances (`T(1,2)` and `T(1,3)`).

We turn datalog into a trust management system by extending the language with SDSI global names [20, 1], which are (local) names paired with public keys. In SD3, we use the syntax `K$E` for the global name of a relation `E` under the control of the keyholder of public key `K`.[2] In SDSI, `K$E` would be pronounced, "`K`'s `E`." Global names can be used in SD3 rules in the same way as local names, for example:

```
T(x,y) :- K$E(x,y)
```

This rule says that `T(x,y)` holds provided that `K$E(x,y)` holds. The SD3 implementation can only conclude that `K$E(x,y)` holds if `K` says that it does, in the sense of BAN logic [10, 2]. For example, we could conclude that `K$E(1,2)` holds if we were given a digital certificate, signed by the private key corresponding to `K`, and asserting `E(1,2)`. In other words, using a global name in SD3 implies authentication.

We also permit global names to come with an IP address, using the syntax `(K@A)$E`. The address can serve as the target of a query for the name. For example, consider these rules:

```
T(x,y) :- (K@A)$E(x,y);
T(x,y) :- T(x,z), T(z,y);
```

As in the first set of rules, these rules specify the transitive closure of a graph. However, in this case the edge relation is not given locally, but can be obtained at a computer with address `A`. A local SD3 evaluator can compute the transitive closure by querying a remote SD3 evaluator at address `A` for `K`'s relation `E`; the result should be the edge relation, contained in a certificate (or certificates) signed by the private key corresponding to the public key `K`. Once the edge relation has been obtained, the rest of the computation can take place locally.

SD3 rules can be used to create "chains of trust":

```
T(x,y) :- (K@A)$G(z), z$E(x,y);
T(x,y) :- T(x,z), T(z,y);
```

In this version of transitive closure, the edge relation can be obtained in two steps. First, an authenticated query/response to `K@A` can obtain the relation `G`; each `z` in `(K@A)$G(z)` should be a key/address pair. Second, each `z` can be queried in turn for an edge relation. The full edge relation is the union of the edge relations of all the `z`s. The rules form chains of trust of length two, with `K` signing the keys `z`, and each `z` signing its edge relation; it is possible to construct arbitrary-length chains of trust in SD3. In this way, we can securely "bootstrap" knowledge of a single, statically configured public key (`K`) into knowledge of many keys (the `z`s), solving the key distribution problem.

[2] We use DSA keys, but we will omit the exact syntax for brevity.

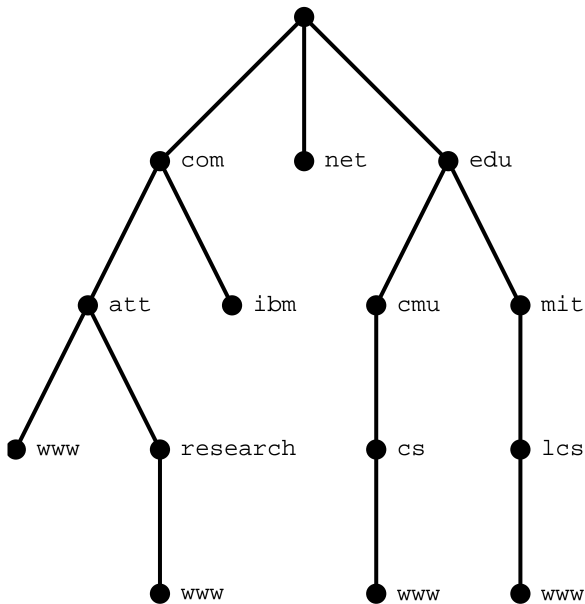

Figure 1. A portion of the domain name space

3 DNS

DNS is the Domain Name System, a distributed database that maps domain names to host data (IP addresses, mailer addresses, and so on). In this section and the next, we give a brief overview of DNS, and show how to implement a secure version of it in SD3. In our overview, we will ignore many important details of the DNS implementation (replication, caching, update, . . .), and instead focus on its overall structure. See Albitz and Liu [4] for a comprehensive treatment.

DNS is structured around the *domain name space*, a tree with labeled nodes (see Figure 1). A *domain* is a subtree of the domain name space. The *name* of a domain is the sequence of labels on the path from the root of the domain up to the root of the entire tree, separated by '.'. For example, the domain name of the lower right node in Figure 1 is 'www.lcs.mit.edu.'. Domain names can also double as the names of hosts.

DNS maps domain names to IP addresses (and other host data). Responsibility for this mapping is divided up among programs called *nameservers*. Each nameserver is associated with a domain. A nameserver can assign addresses to names in its domain, and it can delegate responsibility for subdomains of its domain to other nameservers.

A domain minus its delegated subdomains is called a *zone*. A nameserver only stores data for its zone, and not its complete domain. This zone data is maintained in *resource records* that associate domain names to values. Two types of resource records are of particular interest: address (A) records give the IP address of their associated domain name, and nameserver (NS) records are used to delegate responsibility for subdomains to other nameservers.

Collectively, the zone data of the DNS nameservers defines the complete DNS mapping. However, this data is distributed in the network, and finding the address of a particular name may require zone data from several nameservers. In DNS, a program known as a *resolver* is responsible for querying the different nameservers. Intuitively, a resolver looks up the address of a domain name by querying the root nameserver. If the root nameserver has an A record for the name, it can simply return it. More likely, the root will return an NS record, indicating that it has delegated responsibility for the name to another nameserver. The resolver queries this new nameserver in turn: if the nameserver replies with an A record, the resolver is done; but if the nameserver replies with an NS record, the resolver must query the next nameserver. This continues until an A record is returned, or a nameserver replies with failure.

Thus, to find the address of a name, the resolver gathers a sequence of NS records, starting with the root, and ending with the nameserver that returns an address record for the name: a classic chain of trust. Unfortunately, the chain of trust as implemented in DNS is not secure, because communication between the nameservers is not secure. DNSSEC is an extension of DNS that uses KEY resource records and digital signatures for security. We will show how to specify a core version of DNSSEC in the next section.

4 Specifying DNSSEC in SD3

In Figure 2 we give the SD3 code that implements (a subset of) the zone data of some actual nameservers. The SD3 code for each nameserver begins by giving the IP address where the nameserver is running, and the key that other nameservers can use to authenticate its data. It continues with the rules defining the resource records. Each nameserver maintains a single start-of-authority (SOA) record, associated with its own domain name. The values in the SOA record are the domain that the nameserver is responsible for, and the host name of the nameserver.[3] For example, the root nameserver is called `a.rootservers.net`, and it is in charge of the root (`'.'`) domain. Nameservers use NS records to delegate domains to other nameservers, e.g., the record `NS(com.,a.gtld-servers.net.)` is used by the root nameserver to delegate the `com` domain to the nameserver with name `a.gtld-servers.net`. An address record like `A(a.gtld-servers.net.,198.41.3.38)` gives the IP address associated with a name. Finally, nameservers

[3]Real DNS SOA records also contain some administrative information that we have omitted.

Zone data for the root nameserver

```
site 198.41.0.4; key K1;
SOA(.,a.root-servers.net.) :- ;

NS(com.,a.gtld-servers.net.) :- ;
A(a.gtld-servers.net.,198.41.3.38) :-;
KEY(a.gtld-servers.net.,K2) :- ;
```

Zone data for the com. nameserver

```
site 198.41.3.38; key K2;
SOA(com.,a.gtld-servers.net.) :- ;

NS(att.com.,kcgw1.att.com.) :- ;
A(kcgw1.att.com.,192.128.133.77) :- ;
KEY(kcgw1.att.com.,K3) :- ;

NS(.,a.root-servers.net.) :- ;
A(a.root-servers.net.,198.41.0.4) :- ;
KEY(a.root-servers.net.,K1) :- ;
```

Zone data for the att.com. nameserver

```
site 192.128.133.77; key K3;
SOA(att.com.,kcgw1.att.com.) :- ;

NS(research.att.com.,
   ns.research.att.com.) :- ;
A(ns.research.att.com.,
  192.20.225.4) :- ;
KEY(ns.research.att.com.,K4) :- ;

NS(.,a.root-servers.net.) :- ;
A(a.root-servers.net.,198.41.0.4) :- ;
KEY(a.root-servers.net.,K1) :- ;

A(www.att.com.,192.20.3.54) :- ;
```

Figure 2. DNS resource records in SD3

also have KEY resource records, which associate a public key with names (for brevity, we are using K1, K2, etc., instead of the full keys).

The relation names used by each nameserver are considered local to that nameserver. For example, the root nameserver defines relations named K1$SOA, K1$A, K1$NS, and K1$KEY, and these are distinct from the relations K2$SOA, etc., defined at the other nameservers.

Figure 3 shows a secure resolver written in SD3. The rules define a relation, DNS, between names and addresses: if DNS(n,a) holds, then a is the address of n. The resolver—as is typical in the real DNS—is in fact part of a nameserver (the rules of Figure 3 appear at any nameserver in Figure 2 that wants to offer resolution services), so it has access to the nameserver's zone data. The first rule, which we repeat below, says that if the nameserver has an A record for the name, then the A record determines the mapping:

```
DNS(n,a) :- A(n,a);
```

Otherwise, the resolver should consult another nameserver. There are two subcases. If the resolver's nameserver does not have authority over the name, then the resolver should consult the root nameserver and work down:

```
DNS(n,a) :- SOA(n2,n3), n !>= n2,
            NS(.,n4), A(n4,a4),
            KEY(n4,k), Down(k@a4,n,a);
```

The first line looks at the nameserver's SOA record to see what domain the nameserver is in charge of (n2). If the name we are looking up does not fall in that domain (n !>= n2), then we find the name of the root nameserver (n4), its address (a4), its key (k), and consult the root nameserver using the auxiliary relation Down.

Down is defined so that if Down(x,n,a) holds, then a is the address of n according to the nameserver with key and address given by x. Its definition is simple:

```
Down(x,n,a) :- x$A(n,a);
Down(x,n,a) :- x$NS(n2,n3), n>n2,
               x$A(n3,a3), x$KEY(n3,k),
               Down(k@a3,n,a);
```

The first rule says that Down(x,n,a) holds if x has an address record for n with address a. The second covers the case where x has delegated responsibility for n: x$NS(n2,n3), n>n2 means that n is in the domain n2, which has been delegated to nameserver n3. The rest of the rule looks up the address and key of n3 and consults it, recursively using Down.

The final rule of DNS covers the case where the resolver's nameserver has authority over the name, and it has delegated the name to another nameserver. If so, then it must have an appropriate NS record in its own zone data:

```
DNS(n,a) :- SOA(n2,n3), NS(n4,n5),
            n>n4, n4>n2,
            A(n5,a5), KEY(n5,k),
            Down(k@a5,n,a);
```

```
DNS(n,a) :- A(n,a);

DNS(n,a) :- SOA(n2,n3), n !>= n2,
            NS(.,n4), A(n4,a4), KEY(n4,k), Down(k@a4,n,a);

DNS(n,a) :- SOA(n2,n3), NS(n4,n5), n>n4, n4>n2,
            A(n5,a5), KEY(n5,k), Down(k@a5,n,a);

Down(x,n,a) :- x$A(n,a);

Down(x,n,a) :- x$NS(n2,n3), n>n2,
               x$A(n3,a3), x$KEY(n3,k), Down(k@a3,n,a);
```

Figure 3. A DNSSEC resolver written in SD3

Once the address (a5) and key (k) of the nameserver is known, the lookup proceeds using Down again.

5 Analysis

The policy of our secure DNS resolver is easy to understand, because it is given in less than 10 lines of code. To emphasize this, we give an informal analysis of its security aspects, suggest some quick variations, and discuss their consequences. Keep in mind how difficult it would be to analyze or change the security policy of BIND, working from its source code.

Consider the second rule of our resolver:

```
DNS(n,a) :- SOA(n2,n3), n !>= n2,
            NS(.,n4), A(n4,a4),
            KEY(n4,k), Down(k@a4,n,a);
```

It says that when the name to look up does not come under the authority of the resolver's nameserver, the root nameserver should be consulted. The address of the root nameserver is looked up using A(n4,a4). Since the name A is not qualified, SD3 takes this to refer to K$A, where K is the key of the resolver's nameserver. This means that the resolver will not use some other K'$A(n4,a4') to find the root's address. This is quite important, as every nameserver needs the root address, so there are many different A records for the root, at least one per nameserver. By making this restriction, we limit the number of nameservers in control of the DNS mapping, as determined by the resolver.

More generally, keeping track of global names in our cache helps us prevent cache poisoning [6]. In one form of cache poisoning, a resolver asks a rogue nameserver for records under the authority of the rogue, and the rogue responds with records for domains not under its authority. For example, the rogue might return an A record specifying that the root nameserver has the rogue's own address. The rogue hopes that the resolver will place the record in its cache, and use it the next time it needs to contact the root nameserver. There are two protections in SD3 that prevent this. First, the records returned by the rogue must have valid signatures, or they will not be placed in the cache. Second, the signing principal is entered into the cache along with the records. So the rogue can get a "poison" address record for the root into the cache, provided it signs it with a key under its control; but the rule above will never use the poison record to look up the address of the root (assuming the rogue does not have the resolver's own key).

We could have written the rule another way:

```
DNS(n,a) :- SOA(n2,n3), n !>= n2,
            NS(.,n4), DNS(n4,a4),
            KEY(n4,k), Down(k@a4,n,a);
```

Here we have replaced the lookup of the A record with a recursive reference to the DNS relation. This gives control of the root address to any nameserver in charge of a.root-servers.net. This includes the root nameserver itself, as well as the nameserver for the net subdomain. In fact, this is similar to the actual DNS policy. This can lead to a violation of the principal of least privilege: Bernstein has noted domains whose mappings are potentially controlled by hundreds of nameservers [7].

To repair this and other problems in DNS, Bernstein has suggested that NS records should associate domains with the *addresses* of nameservers, rather than their names. This would eliminate the DNS lookup on the name of the nameserver. Making this change to the real DNS is impractical, but, it is easy to experiment with in SD3. First, we introduce a new NS2 resource record, associating a domain with the address and key of the nameserver in charge of that domain. For example, the NS2 record for the root would be

```
NS2(.,198.41.0.4,K1) :- ;
```

We can then modify the rules of the resolver to use the new resource record, for example,

```
DNS(n,a) :- SOA(n2,n3), n !>= n2,
            NS2(.,a4,k), Down(k@a4,n,a);
```

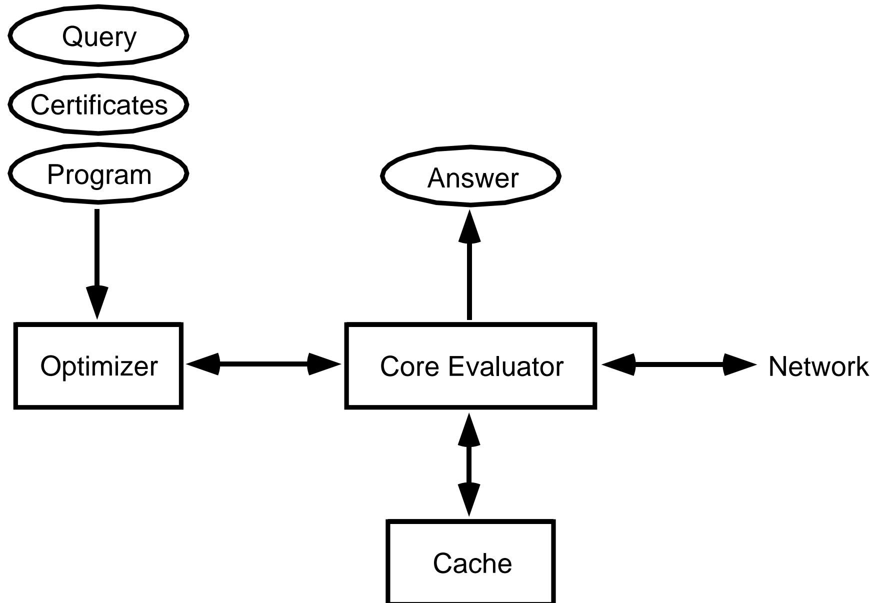

Figure 4. The SD3 evaluator

The change to the resolver can be accomplished by replacing one 10 line program with another.

Finally, we consider a kind of denial of service attack which can be caused by a misconfigured or corrupted nameserver. Suppose an attacker registers `bad.com`, so it controls the nameserver, `N`, for that domain. The nameserver `a.gtld-servers.net` for `com` will delegate `bad.com` to that nameserver: it will have a record `NS(bad.com.,N)`. The attacker could now delegate `bad.com` back to `a.gtld-servers.net`, using a record `NS(bad.com.,a.gtld-servers.net)`. Then, if the attacker can get someone to look up the domain `bad.com`, their resolver may go into a loop.

SD3, on the other hand, will detect the loop immediately and halt, with the correct answer (none). This is because recursion is built in to SD3, and, indeed, is necessary for realistic policies. For example, our rules for `Down` are recursive; and recursion is needed for *mutual trust*, e.g., where Alice trusts Bob and delegates to him, and vice versa. So, SD3 has been built from the start with loop detection.

6 Certified evaluation

The structure of our evaluator is shown in Figure 4. It is a modified datalog interpreter with three parts: an optimizer; a cache; and a core evaluator. Evaluation is initiated by sending a query, input certificates, and the local program through the optimizer. This produces an optimized set of rules that is passed to the evaluator, which places them in its cache. The core evaluator does a standard datalog evaluation on the cache, except that it is capable of detecting when queries need to be sent out into the network. The evaluator expects replies to be certificates containing rules; after verifying the certificates, the rules of the response are passed through the optimizer, producing an optimized set of rules that is placed in the cache. Eventually, the evaluator returns the answer to the query.

Datalog has been studied for several decades, so in implementing SD3 we have been able to incorporate standard, off-the-shelf database evaluation techniques, rather than reinventing them (see [3] for a sampling). We have written a separate paper that discusses the more novel implementation techniques of our evaluator, as well as the theoretical foundations of SD3 [18].

Perhaps the most notable feature of our evaluator from the security perspective is that in addition to producing answers to queries, it also produces a proof that the answers are correct: it is a *certified* evaluator. The proof is run through a very simple checker before the answer is reported by the evaluator. If the proof does not check, the evaluator reports an error. Because the checker is so simple, this gives us a high assurance that the answers produced by the evaluator are correct, despite its more complex implementation. In the rest of this section, we describe the proofs produced by the evaluator and the operation of the checker.

The proof produced by the evaluator shows that a list of facts (the answer to the query) follow from the input security policy. It consists of a list of initial facts, a list of rules, a list of instructions on how to use the rules to deduce facts,

Assumptions

```
0.  K3$SOA(att.com.,kcgw1.att.com.)
1.  K3$NS(.,a.root-servers.net.)
2.  K3$A(a.root-servers.net.,198.41.0.4)
3.  K3$KEY(a.root-servers.net.,K1)
4.  K1$A(a.gtld-servers.net.,198.41.3.38)
```

Rules

```
0.  K3$DNS(n,a) :- K3$SOA(n2,n3), n !>= n2,
                   K3$NS(.,n4), K3$A(n4,a4),
                   K3$KEY(n4,k),
                   K3$Down(k@a4,n,a);
1.  K3$Down(x,n,a) :- x$A(n,a);
```

Instructions

```
1: 4
0: 0,1,2,3,5
```

Results

```
6: K3$DNS(a.gtld-servers.net.,198.41.3.38)
```

Figure 5. A proof of `K3$DNS(a.gtld-servers.net.,198.41.3.38)`

and a list of indices that specify which of the deduced facts correspond to the answer. We will illustrate proofs and the operation of the checker by example.

Suppose we query the `att.com` nameserver from Figures 2 and 3 for the address of `a.gtld-servers.net`: the query is

```
K3$DNS(a.gtld-servers.net.,x)
```

and the answer is

```
K3$DNS(a.gtld-servers.net.,198.41.3.38)
```

The evaluator can produce this answer after a single query/response with the root nameserver. At the same time, it produces a proof for the answer, shown in Figure 5. The initial facts come from rules with no conditions: they always hold. In this case, 5 facts were used, numbered 0 to 4. Notice that all the facts use fully qualified global names (we've omitted the IP addresses from these names, for brevity). Most of the facts are qualified with `K3`, which is the key of the local nameserver. However, fact 4 is qualified with `K1`, the key of the root nameserver. We permit this as a *fact* because it comes from a certificate signed by `K1`, obtained by the query/response. We permit it as an *initial* fact because the checker is not allowed to do any query/response of its own. This is why the proof must contain fully qualified names: SD3 does not treat address records signed by `K1` and `K3` the same, and the checker must make sure of the difference.

The rules are a subset of the local rules; the evaluator only lists the rules that were needed to obtain the answer. In this case, two rules were used, numbered 0 and 1. Once again, the rules use fully qualified names.

The instructions, which are not numbered, consist of a rule number, followed by a colon, followed by a list of fact numbers. An instruction tells the checker to apply the rule using the facts from left to right, to obtain a new fact. For example, the first instruction, `1: 4`, says to apply rule 1 using fact 4. Following this instruction, the checker deduces

```
K3$Down(K1,
        a.gtld-servers.net.,
        198.41.3.38)
```

This becomes fact number 5. The final instruction says to apply rule 0 using facts 0, 1, 2, 3, and the new fact 5, in order. The checker deduces

```
K3$DNS(a.gtld-servers.net.,198.41.3.38)
```

which becomes fact 6.

After all of the instructions have been followed, the checker has to verify that the answers have been derived from the rules. The results section simply lists the answers and their position in the list of facts. In this case, there is a single answer, `K3$DNS(a.gtld-servers.net.,198.41.3.38)`, and the results section says that it should match fact 6. Once the checker does this simple verification, it can report that the proof is valid.

Clearly, the process of checking is quite simple, because the proof contains such detailed instructions for finding the answer from the rules. In fact, our checker is currently written in less than 100 lines of code.

The job of the evaluator is much harder, however. It has the local rules to work with, but it does not have instructions on which rules to apply, and which facts to use; it has to deduce this itself. It also has to decide when to query for remote certificates. And finally, it has to produce the proof for the checker. This is more difficult than it might seem, because our optimizer works by rewriting the local rules, based on the input query, into a more efficient form that is passed to the core evaluator. So, the evaluator is not even working from the same set of rules as must appear in the proof. The optimizer and evaluator must be coordinated to produce a proof in terms of the original, unoptimized rules.

Certified evaluation thus reduces the trusted computing base from a large and complex evaluator to a small and simple checker. It is still possible for the evaluator to produce a wrong answer, but it can never produce a proof for that wrong answer that will pass the checker. It is also possible for the evaluator to produce a correct answer but an incorrect proof. In either case, the checker will report failure, indicating a definite bug in the evaluator; not only does the checker prevent incorrect answers, it essentially acts like an assert statement, helping us to debug the evaluator.

7 Related work

SD3 is a successor to the QCM trust management system [13, 15, 14, 19]. Like SD3 and some of the other trust management systems we will discuss, QCM uses logic to encode security policies. The innovation of QCM was to notice that many security policies can be expressed almost entirely within the fragment of logic that forms the core of database programming languages, hence enabling automatic certificate retrieval (or, more generally, credential retrieval). We believe that automatic certificate retrieval can and should be applied to other trust management systems (e.g., SPKI [12], KeyNote [8]). Variations of QCM have been based on the relational algebra [13], set comprehensions [15], and now, with SD3, datalog—all of which are roughly equivalent, by variations of Codd's Theorem. SD3 extends QCM with recursive policies (which were important in our DNSSEC example, and which also arise in cases of mutual trust) and certified evaluation.

The importance of linked, authenticated namespaces in trust management was pointed out by Rivest and Lampson's SDSI [20]. We introduced them in Section 2 by the example

```
T(x,y) :- (K@A)$G(z), z$E(x,y);
```

Here `(K@A)$G` is the name for a relation `G` authenticated by `K`, and, more importantly, the key `z` *extracted from* `(K@A)$G` is used to form the authenticated name `z$E`, whose contents can then be securely accessed. This is much like extracting a hyperlink from a web page and following it. Linked names are a crucial security mechanism, because they allow knowledge of one key to be bootstrapped securely into knowledge of many keys; surprisingly, not all trust management systems have linked names. We feel that SD3 can express roughly the same policies as SDSI 2, which has merged with SPKI [12]; however, we cannot make a definitive statement, because SPKI's tag intersection operation is not specified. SDSI and SPKI do not do certificate retrieval.

KeyNote [8] is a trust management system that seems to be about as expressive as SDSI/SPKI. In place of SPKI's tag intersection, KeyNote provides a variety of arithmetic and comparison operators for use in policies. When SD3 is equipped with these same operators, we can express KeyNote policies. KeyNote does not have automatic certificate retrieval.

PolicyMaker [9] is the system that introduced the term "trust management." PolicyMaker can express policies that SD3 cannot, because its policies can rely on programs in arbitrary safe languages. PolicyMaker is thus a universal language, while the core of SD3, like datalog, lies in P. The limited expressiveness of SD3 helps make automatic certificate retrieval possible; PolicyMaker does not retrieve certificates.

REFEREE [11] is a trust management system in which the policy writer can write policies that cause certificates to be retrieved. That is, a REFEREE policy can retrieve certificates, but it must be done explicitly. In SD3, the evaluator decides when to retrieve certificates, automatically; instructions to retrieve certificates do not appear in SD3 policies, so, they are shorter and easier to understand and write.

Oasis [16] is a distributed access control system; we consider it to be a trust management system. Oasis is based on Horn clauses, and, therefore, is closely related to SD3. Oasis seems to lack linked names, and does not have automatic certificate retrieval; clients are responsible for gathering certificates. Oasis supports revocation in an interesting way: the issuer of a certificate keeps track of what other certificates were used as evidence that the first certificate should be issued. Anyone who relies on a certificate notifies the issuer, and the issuer will notify them if a certificate is revoked. Revoking a certificate can thus cause the revocation of other certificates that rely on it. The system essentially maintains a proof tree for each issued certificate. In contrast with the proofs constructed by SD3's certified evaluator, a proof tree in Oasis is distributed in pieces in the network, and is used for a different purpose (revocation rather than reducing the trusted computing base).

Trust Policy Language (TPL) [17] is a trust management system with automatic certificate retrieval. The authors show that the monotonic portion of TPL can be translated to Prolog; by the same technique, we can build a translation to SD3. TPL does not use linked names, and uses a different strategy than SD3 for determining where to retrieve certificates.

Delegation Logic (DL) [21] is a trust management system based on a logic programming language. DL does not provide certificate retrieval, and permits non-monotonic policies (SD3 is strictly monotonic).

Certified evaluation has similarities to several existing techniques. A *certified compiler* is a compiler that produces object code from source code, and at the same time, produces a proof that the object code satisfies certain safety policies [23]. The proof can be run through a simple checker. *Translation validation* is another compiler technique, in which the compiler produces not a safety proof for the object code output, but rather, a proof that it is semantically equivalent to the source [22].

Appel and Felten's *proof-carrying authentication* [5] formalizes several authentication frameworks in logic, thus reducing authentication tests to checking whether a logical statement is valid. To simplify the job of the resource manager, a client wanting to use a resource is required to produce a proof of the logical statement which can be checked very simply. These proofs are similar to the proofs produced by SD3's certified evaluator, but SD3 uses them for a slightly different purpose: we want to ensure that the an-

swers produced by the evaluator are correct. Therefore, our checker is part of the evaluator, and the proof never travels over the network (though there is nothing to prevent this if it is desired). In contrast, the Appel/Felten system only has a checker, and the proof comes across the network. The logic supported by Appel and Felten's system is more powerful than SD3. However, SD3 has advantages. First, not only does it include a checker, but its certified evaluator produces proofs automatically, instead of by hand, as is currently the case in the Appel/Felten system. Second, SD3 is capable of retrieving certificates for use in its proofs, while the Appel/Felten system does not do certificate retrieval.

The Kimera project [25, 24] tests Java bytecode verifiers and virtual machines by feeding them millions of automatically generated class files. If two verifiers or JVMs disagree on the same input, then there is a bug in one of them. These duelling verifiers and JVMs are similar to our evaluator and checker, except that one of the duellists (the evaluator) gives considerable help to the other.

8 Conclusion

We have described SD3, a trust management system consisting of a high-level policy language, a local policy evaluator, and a certificate retrieval system. We have shown that SD3 policies are easy to write and understand, and our version of DNSSEC shows that SD3 can be used to build realistic security infrastructures.

A unique feature of SD3 is its certified evaluator: as the evaluator computes the answer to a query, it also computes a proof that the answer follows from the security policy. Before the answer is returned, the proof is passed through a simple checker, and incorrect proofs are reported as errors. The certified evaluator reduces the trusted computing base and greatly increases our confidence that the answers produced by the evaluator follow from the specification, despite complex optimizations.

References

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

[2] Martín Abadi, Michael Burrows, Butler Lampson, and Gordon D. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, September 1993.

[3] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995.

[4] Paul Albitz and Cricket Liu. *DNS and BIND*. O'Reilly, third edition, 1998.

[5] Andrew W. Appel and Edward W. Felten. Proof-carrying authentication. In *6th ACM Conference on Computer and Communications Security*, November 1999.

[6] Steven M. Bellovin. Using the domain name system for system break-ins. In *Proceedings of the 5th USENIX UNIX Security Symposium*, 1995.

[7] D.J. Bernstein. Notes on the domain name system. Internet publication at http://cr.yp.to/djbdns/notes.html.

[8] Matt Blaze, Joan Feigenbaum, and Angelos D. Keromytis. KeyNote: Trust management for public-key infrastructures. In *Cambridge 1998 Security Protocols International Workshop*, England, 1998.

[9] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *Proceedings of the 17th Symposium on Security and Privacy*, pages 164–173. IEEE Computer Society Press, 1996.

[10] Michael Burrows, Martín Abadi, and Roger M. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8(1):18–36, February 1990.

[11] Yang-Hua Chu, Joan Feigenbaum, Brian LaMacchia, Paul Resnick, and Martin Strauss. REFEREE: Trust management for web applications. *The World Wide Web Journal*, 2(3):127–139, 1997.

[12] Carl M. Ellison, Bill Frantz, Ron Rivest, and Brian M. Thomas. SPKI certificate documentation. http://www.clark.net/pub/cme/html/spki.html.

[13] Carl A. Gunter and Trevor Jim. Design of an application-level security infrastructure. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, September 1997.

[14] Carl A. Gunter and Trevor Jim. Generalized certificate revocation. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 316–329, January 2000.

[15] Carl A. Gunter and Trevor Jim. Policy-directed certificate retrieval. *Software: Practice & Experience*, 30(15):1609–1640, September 2000.

[16] R.J. Hayton, J.M. Bacon, and K. Moody. Access control in an open distributed environment. In *IEEE Symposium on Security and Privacy*, pages 3–14, May 1998.

[17] Amir Herzberg, Yosi Mass, Joris Michaeli, Dalit Naor, and Yiftach Ravid. Access control meets public key infrastructure, or: Assigning roles to strangers. In *IEEE Symposium on Security and Privacy*, May 2000.

[18] Trevor Jim and Dan Suciu. Dynamically distributed query evaluation. In *Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, Santa Barbara, California, May 2001.

[19] Pankaj Kakkar, Michael McDougall, Carl A. Gunter, and Trevor Jim. Credential distribution with local autonomy. In *The Second International Working Conference on Active Networks*, October 2000.

[20] Butler Lampson and Ron Rivest. SDSI—a simple distributed security infrastructure. http://theory.lcs.mit.edu/~cis/sdsi.html.

[21] Ninghui Li, Benjamin Grosof, and Joan Feigenbaum. A practically implementable and tractable delegation logic. In *IEEE Symposium on Security and Privacy*, May 2000.

[22] George C. Necula. Translation validation for an optimizing compiler. In *Proceedings of the 2000 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI00)*, Vancouver, British Columbia, Canada, June 2000.

[23] George C. Necula and Peter Lee. The design and implementation of a certifying compiler. In *Proceedings of the '98 Conference on Programming Language Design and Implementation*, Montreal, 1998.

[24] Emin Gün Sirer. Testing Java virtual machines. In *International Conference on Software Testing And Review*, San Jose, California, November 1999.

[25] Emin Gün Sirer, Sean McDirmid, and Brian N. Bershad. Verifying verifiers. In *Workshop on Security and Languages*, Palo Alto, September 1998.