Binder, a Logic-Based Security Language

John DeTreville
Microsoft Research
johndetr@microsoft.com

Abstract

We introduce the concept of a security language, *used to express security statements in a distributed system. Most existing security languages encode security statements as schematized data structures, such as ACLs and X.509 certificates. In contrast,* Binder *is an open logic-based security language that encodes security statements as components of communicating distributed logic programs. Binder programs can be more expressive than statements in standard security languages, and the meanings of standard security constructs and operations such as certificates and delegation are simplified and clarified by their formulation in Binder. Translation into Binder has been used to explore the design of other new and existing security languages.*

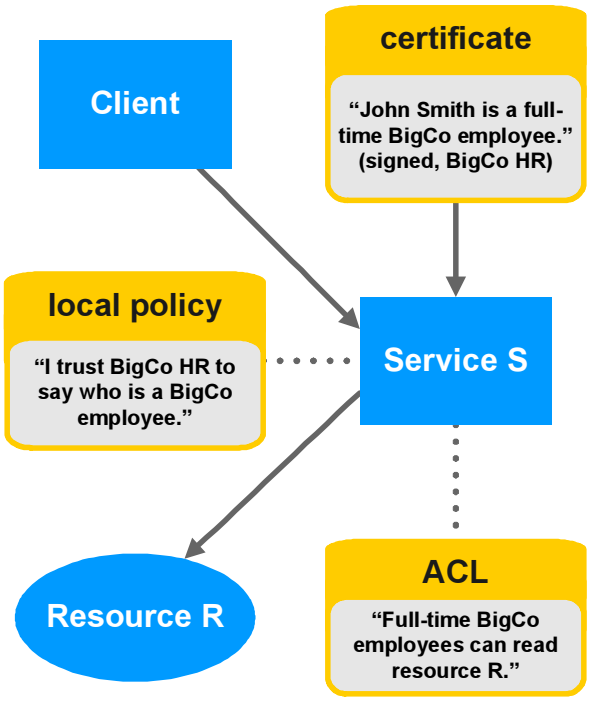

Figure 1. Certificates, policies, and ACLs

1. Security languages

Access control decisions in a loosely-coupled distributed environment are driven by distributed *security statements.* As shown in the example in Figure 1, these statements can be stored in a variety of places: in signed *certificates* that can flow among the parties; in *policies* local to the services; in *access control lists (ACLs)* associated with the individual resources; and perhaps elsewhere. When a client requests an operation on a resource, the service controlling that resource—here, service S controls resource R—uses the security statements available to it to determine whether to grant or deny the requested access. In this example, service S would presumably allow John Smith to read resource R.

Traditional systems store security statements in a variety of data structures. The certificate shown here might be an X.509 certificate that attests to an identity [12]; the local policy might enumerate the X.509 roots that the service will trust to certify identities; and the ACL might be an ordered list of pairs that map users' identities to their access rights. A predefined decision procedure matches these data structures against the identity of any client requesting an operation, thereby verifying the client's access rights.

However these security statements are encoded, they must necessarily obey some formal schema. We can say that this schema and its accompanying decision procedure define a *security language,* and that our certificates, policies, ACLs, etc., are formed from security statements written in our security language and interpreted by its decision procedure. For example, since X.509 specifies the form and meaning of X.509 certificates, X.509 is a security language. SDSI and SPKI are other security languages, as are PolicyMaker and KeyNote, and so on.

Proceedings of the 2002 IEEE Symposium on Security and Privacy (S&P'02)
1081-6011/02 $17.00 © 2002 **IEEE**

Many existing security languages are designed for very specific domains—like X.509, meant to control access to an X.500 database—and each can express some statements more readily than others. X.509 excels at building chains of Certification Authorities (CAs). SDSI lets us define and refer to principals and groups of principals (e.g., the group of all company employees). PolicyMaker is a language for encoding a service's local security policy. Inevitably, in any given domain, some security languages are more expressive than others.

If we are designing a closed system with known requirements, we may be able to choose a minimalist security language, closely matching its design to our needs. Conversely, if we are designing an open system that will be used in unexpected ways and that will evolve in unknown directions, then it might be better to make our language more expressive than currently needed.

This paper presents the design of a new logic-based security language for open systems—called *Binder*—that is intended to be more expressive than most existing security languages, while remaining practical. Binder does not directly implement higher-level security concepts like delegation, but provides flexible low-level programming tools to do so. Our experience with Binder suggests that logic programming can be a useful foundation for a practical security language, and that it can also help us explore new and existing security languages. The section below on related work draws more specific comparisons with existing security languages.

2. Datalog for authorization

Binder is an extension of the datalog logic-programming language, which can be decided in polynomial time [17]. (Datalog is a restricted subset of the well-known Prolog logic-programming language [15].) An EBNF grammar of Binder may be found in Appendix A. Binder extends datalog with constructs for communicating securely across a distributed environment, but we use the datalog subset of Binder in this section to write local security programs that do not communicate.

Let us imagine that John Smith wishes to read resource R. By convention, we will grant this access if and only if we can derive the authorization atom

```
can(john_smith, read, resource_r)
```

(An atom combines a predicate and one or more terms. Here, `can` is a predicate and `john_smith`, `read`, and `resource_r` are constant terms.) A simple ACL for resource R might be represented by the (tedious) datalog program

```
can(john_smith, read, resource_r).
can(john_smith, write, resource_r).
can(fred_jones, read, resource_r).
…
```

at service S. (Statements of this form, with a single atom—a single predicate applied to zero or more terms—are called *facts*.) Since our authorization atom is part of this program, it is trivially derivable and access is granted.

To raise the level of allowable abstraction, existing security languages like SDSI also let us define groups of principals (like John Smith and Fred Jones). We can also model groups in datalog, as in the different datalog program

```
can(X, read, resource_r) :-
  employee(X, bigco).
employee(john_smith, bigco).
…
```

The first statement is a *rule* stating that principal X—a variable term—can read resource R if X is a BigCo employee; the atom on the left is derivable if the atom or atoms to the right also are. (Variables begin with uppercase letters, while constants begin with lower-case letters.) The second statement is a fact, stating that John Smith is a BigCo employee. Again, our authorization atom is derivable with `X=john_smith`, and access is granted.

While datalog can express abstractions that are also expressible in existing security languages, like groups, it can express more powerful and more general concepts too. Consider the following datalog program.

```
can(X, read, resource_r) :-
  employee(X, bigco),
  boss(Y, X),
  approves(Y, X, read, resource_r).
employee(john_smith, bigco).
boss(fred_jones, john_smith).
approves(fred_jones, john_smith,
         read, resource_r).
…
```

The first statement is a *rule* stating that principal X can read resource R if X is a BigCo employee and X's boss (Y) approves. Using new predicates, datalog lets us define and use new relations as needed to express our desired security policies. In contrast, SDSI's existing mechanism for defining groups is not powerful enough to model this example policy.

Datalog programs can encode a wide range of security policies, but an open distributed system with multiple administrative domains will have multiple interoperating policies. It is no more practical to encode these various interoperating policies in a single datalog program than it would be to encode them in a single global database. (What single party could maintain the program or the database? How would everyone agree?) Instead, Binder lets separate programs (separate databases) interoperate correctly and securely.

Figure 2. Communicating contexts

3. Communicating contexts

Each component of a distributed environment has its own local Binder *context* with its own Binder program, where certain local Binder atoms are derivable. A service uses its local Binder context to make its local authorization decisions, and Binder provides extensions to datalog for these distributed contexts to work together.

Binder contexts communicate via signed *certificates,* as shown in Figure 2. Each Binder context has its own cryptographic key pair; the exporting context uses the private key (which it keeps secret) to sign statements, and the corresponding public key—used to verify the signature at the importing context—also serves to name the context, as in SDSI/SPKI.

A statement from one Binder context—fact, rule, or derivable atom—may be *exported* into a signed certificate, and later *imported* from the certificate into another context. Imported statements are automatically quoted using `says` to distinguish them from local assertions. If the public key `rsa:3:c1ebab5d` belongs to BigCo HR—real keys are much longer, of course—then the statement

```
employee(john_smith, bigco)
```

exported by BigCo HR would be imported as

```
rsa:3:c1ebab5d says
  employee(john_smith, bigco).
```

(Appendix B contains a more precise explanation of the rules for importing statements.) If the importing context has a rule like

```
employee(X, bigco)
  :- rsa:3:c1ebab5d says
       employee(X, bigco).
```

then `employee(john_smith, bigco)` is also derivable there. In the absence of any such rule, the imported statement will by default be inert and will not take further part in the decision procedure.

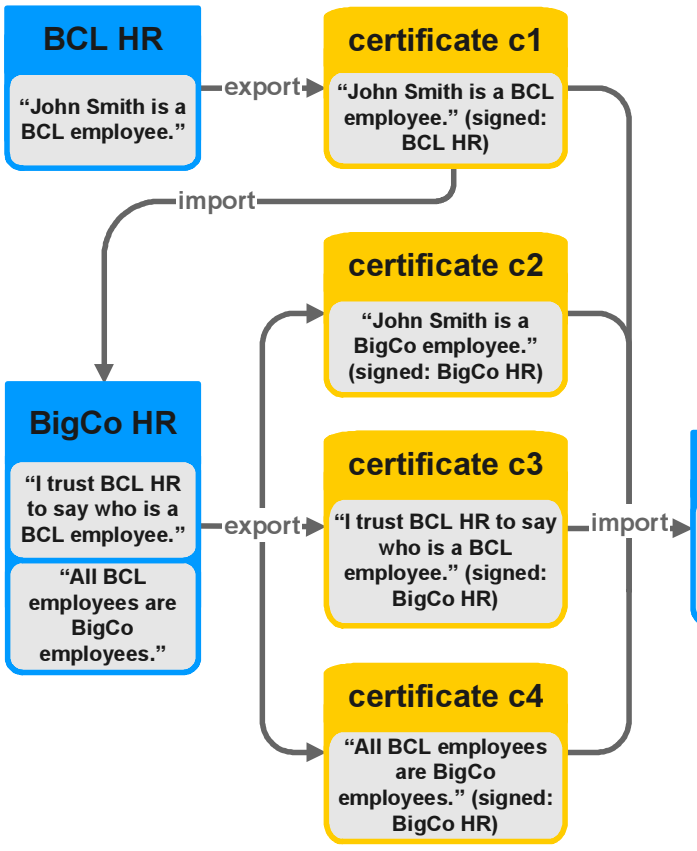

Figure 3: Possible certificate flow

4. Delegation and trust

In Binder, statements from any Binder context may be exported and later imported. Since imported statements are automatically quoted with `says`, the local context can treat imported statements differently from local statements. The controlled importation of signed statements is Binder's mechanism for "trust" (as in, "Service S trusts BigCo HR") or "delegation" ("Service S delegates the identification of BigCo employees to BigCo HR") or "speaks-for" ("BigCo HR speaks for service S"); Binder lets us implement an unambiguous logic-based policy with the same effect.

Let's extend the example from Figure 1 by adding an additional level of indirection. In Figure 3, BigCo HR has delegated the identification of BigCo Labs (BCL) employees to BCL HR, and all BCL employees are BigCo employees. Our goal is still to convince service S that John Smith is a BigCo employee, but the necessary information can flow along multiple distinct paths in different scenarios.

In one scenario, BCL HR exports certificate c1 to BigCo HR, whose local policy allows its import. BigCo

HR now concludes that John Smith is a BigCo employee, and exports certificate c2 to service S, whose local policy allows its import. Service S now concludes that John Smith is a BigCo employee.

Alternatively, BCL HR can export certificate c1 directly to service S, and BigCo HR can export certificates c3 and c4 also directly to service S, which can now conclude, as above but on its own, that John Smith is a BigCo employee. Here, we model a traditional "chain of trust": service S trusts BigCo HR to establish a policy, while BigCo HR trusts BCL HR.

5. Example of Binder programs

This section shows the complete Binder programs for the examples from Figures 1 and 3. Here, BigCo HR's public key is `rsa:3:c1ebab5d`, while BCL HR's public key is `rsa:3:8e72145b`.

5.1 Example from Figure 1

Program 1 shows the English security statements from Figure 1 and their translations into Binder.

#	English statement	Binder statement
1a	"John Smith is a full-time BigCo employee"	`employee(john_smith, bigco, full_time).` *(original form, in the context of BigCo HR)*
1b		`rsa:3:c1ebab5d says employee(john_smith, bigco, full_time).` *(as imported into the context of service S)*
2	"I trust BigCo HR to say who is a BigCo employee"	`employee(X, bigco, S) :- rsa:3:c1ebab5d says employee(X, bigco, S).`
3	"Full-time BigCo employees can read resource R"	`can(X, read, resource_r) :- employee(X, bigco, full_time).`
4	"John Smith can read resource R"	`can(john_smith, read, resource_r).`

Program 1. English statements from Figure 1 and their translations into Binder

Statement 1a— "John Smith is a full-time BigCo employee"—is shown in the context of BigCo HR; while statement 1b is shown after it has been imported into the context of service S. Statement 1b is explicitly quoted as coming from BigCo HR (`rsa:3:c1ebab5d`).

Statement 2 shows the establishment of trust in statements from BigCo HR; variable S stands for the employment status (e.g., `full_time`). If BigCo HR's public key appears often in our program, we might choose to write

```
employee(X, bigco, full_time) :-
  Y says employee(X, bigco, full_time),
  bound(bigco_hr, Y).
bound(bigco_hr, rsa:3:c1ebab5d).
```

and bind the local name `bigco_hr` to a public key. We can even refer to local names elsewhere on the distributed system, simulating the linked name spaces of SDSI/SPKI, but without built-in language support.

In this example, names like `john_smith`, `bigco`, and `full_time` pass unchanged from BigCo HR to service S; more complex mappings can be implemented by additional rules, perhaps carrying along extra public keys to root these names as in SDSI/SPKI. For example, we might explicitly write

```
employee(
  rsa:3:c1ebab5d, john_smith,
  rsa:3:c1ebab5d, bigco,
  rsa:3:c1ebab5d, full_time)
```

to associate these names with a particular name space, while modifying the other rules accordingly.

Finally, statement 3 shows the statement "Full-time BigCo employees can read resource R," while statement 4 shows the derived atom at service S that gives John Smith access to resource R.

5.2 Example from Figure 3

Program 2 shows the English security statements from Figure 3 and their translations into Binder.

#	English statement	Binder statement
1a	"John Smith is a BCL employee"	`employee(john_smith, bcl).` *(original form, in the context of BCL HR)*
1b		`rsa:3:8e72145b says employee(john_smith, bcl).` *(as imported into the context of BigCo HR or service S)*
2a	"I trust BCL HR to say who is a BCL employee"	`employee(X, bcl) :- rsa:3:8e72145b says employee(X, bcl).` *(original form, in the context of BigCo HR)*

2b		`rsa:3:c1ebab5d says employee(X, bcl) :- rsa:3:8e72145b says employee(X, bcl).` *(as imported into the context of service S)*
3a	"All BCL employees are BigCo employees"	`employee(X, bigco) :- employee(X, bcl).` *(original form, in the context of BigCo HR)*
3b		`rsa:3:c1ebab5d says employee(X, bigco) :- rsa:3:c1ebab5d says employee(X, bcl).` *(as imported into the context of service S)*
4	"I trust BigCo HR to say who is a BigCo employee"	`employee(X, bigco) :- rsa:3:c1ebab5d says employee(X, bigco).`
5a	"John Smith is a BigCo employee"	`employee(john_smith, bigco).` *(in the context of BigCo HR)*
5b		`rsa:3:c1ebab5d says employee(john_smith, bigco).` *(in the context of service S, after certificate import or local derivation)*
5c		`employee(john_smith, bigco)` *(in the context of service S, after further local derivation)*

Program 2. Security statements from Figure 3 and their translations into Binder

Statement 1—"John Smith is a BCL employee"—is shown in its original form at BCL HR and as imported into either BigCo HR or service S.

Statement 2 shows the establishment of trust in statements from BCL HR, both at BigCo HR and as imported into service S. Note that statement 2b has been rewritten from its expected form; this is discussed in detail in Appendix B.

Statement 3 is shown at BigCo HR and at service S. Statement 4 is shown at service S.

Statement 5 is shown in multiple forms because of the different certificate flows possible. Statement 5a can be derived at BigCo HR and imported into service S as statement 5b; statement 5b can also be derived directly at service S using statements 1b and 2b; statement 5c can be derived at service S using statements 4 and 5b.

6. Proofs, monotonicity, and revocation

A service grants access to a resource in Binder only when it can derive an atom saying it should; otherwise, by default, access is denied. The derivation steps form a *proof* that access should be granted.

A proof can be generated at the service—as traditionally—or we can require that the client generate the proof and transmit it with the request. If so, the service need only check the proof; this optimization can offload work from a heavily loaded service onto its less busy clients, while also helping avoid denial-of-service attacks. (This approach is also used by Jim [13] and by Appel and Felten [3].) Since the service's policy is stored as a Binder program, and since Binder statements can be passed in certificates, the service can pass its policy to the client in preparation for the construction of such a proof.

Binder is *monotonic*—if an atom is derivable, it's still derivable if we add more statements [15]. Monotonicity is appropriate in a distributed environment, since withholding some statements from a service will not cause it to grant greater access rights. Moreover, a proof generated on a client with little information available will still check on a service with more information.

One consequence of monotonicity is that traditional certificate revocation cannot be modeled from inside Binder; it requires additional mechanism. We have studied three ways to extend Binder to support revocation reliably.

One is through *short-lived statements.* We can attach validity intervals to each Binder statement, as with traditional certificates, and constrain the validity intervals of derived atoms accordingly. Once a statement expires, it can be removed from all contexts, along with all atoms that cannot be derived without it.

A second approach is through a language extension allowing *freshness constraints* on statements. If a derivation rule depends on `fresh P(X, Y)`, say, instead of just `P(X, Y)`, then a new `P(X, Y)` must be derived for each use. This may involve contacting the exporters of old certificates to obtain fresher ones. A generalization of this mechanism is to allow each use of a certificate to specify how fresh it must be.

The final approach is to reference distributed *state.* For example, a statement could have an associated Boolean state "valid" that turns from *true* to *false* if it is revoked. This state could be explicitly referenced from Binder, perhaps with a freshness constraint. Such support for state, while problematical, might also be needed for Binder to emulate features of digital rights management languages as discussed below.

If the validity of a proof can vary with time, a proof that checks at a client may not check at the service. If so, the client can be informed of its error—e.g., that a particular statement is no longer fresh enough—and asked to regenerate the proof.

7. Taxonomy and related work

The Binder security language has five key properties.

1) A statement in Binder can be translated into a declarative, stand-alone English sentence. This is known good practice for messages in a security protocol [1] and we propose that it is even better practice for statements in a security language.
2) Binder programs can explicitly define new, application-specific predicates, which can act as lemmas in proofs. Predicates can be defined recursively. Rich proofs are allowed.
3) Certificates can contain arbitrary statements, including definitions and uses of new application-specific predicates. These certificates can be safely interpreted outside their exporting context.
4) Binder statements can appear in certificates, in policies, in ACLs, and elsewhere, and these statements can interoperate freely.
5) Queries in Binder are decidable in polynomial time, as outlined in Appendix C.

None of the existing languages compared below—X.509, SDSI/SPKI, PolicyMaker and KeyNote, SD3 and other logic-based security languages, and various digital rights management (DRM) languages—shares all of these properties. With a few exceptions, we believe that Binder provides functionality as great as any of these languages and is more appropriate for use in open systems.

7.1 X.509

An X.509 certificate is a signed n-tuple, where n is large and most of the fields are optional. This n-tuple can be thought of as asserting a predicate $P(x_1, x_2, x_3, \ldots, x_n)$ over the values it contains, but X.509 certificates have no straightforward way to say *which* P is being used. (Thus, the translation of an X.509 certificate into English has no verb. Perhaps the predicate is best thought of as the constant *is_an_X509_certificate*.) X.509 thus does not share properties 1–3. X.509 also fails property 4; it can be used only in certificates, not in policies or ACLs.

A complex X.509 certificate may often be factored into a number of smaller Binder certificates, rather like a translation from a CISC architecture to a RISC architecture; the operations may require more steps but these individual steps can combine in more ways. The access control decisions in Binder programs are more explicit than in X.509, and perhaps more understandable in many cases.

In X.509 it is easy to talk about a security decision requiring the approval of one of a certain class of CAs, but hard to talk about the approval of *k*-out-of-*n* CAs. This is because X.509 depends so directly on the construction of linear chains of certificates.

Much of the difficulty in using X.509 comes from its great complexity and many implicit mechanisms [9]. We can expect that a simpler, more explicit language like Binder might be easier to use as well as more expressive.

7.2 SDSI and SPKI

SDSI/SPKI programs do not explicitly encode the predicate being defined. Instead, SDSI statements build their meaning from an implicit "speaks-for" predicate [16, 2], while SPKI also encodes the predicate into the "tags" in SPKI statements [8]. Nevertheless, SDSI/SPKI statements can be translated directly into English. While SPKI programs can define multiple predicates, SDSI programs can define only the speaks-for predicate, and thus SDSI does not share properties 2 and 3. Even SPKI cannot define arbitrary predicates: the `boss` example in Section 2 cannot easily be defined in SPKI, since the tags cannot contain (i.e., be parameterized by) constrained variables like `Y`.

Formalizing SDSI's speaks-for relationship is difficult [10], and Binder does not attempt to do so. Instead, much the same effect is achieved using explicit rules in the Binder language, as in the "trust" statements in Programs 1 and 2.

Delegation is represented clumsily in SPKI. If the local Department of Motor Vehicles (DMV) is to be authorized to license drivers, then the DMV must itself be a licensed driver. Binder's explicit handling of delegation avoids such problems.

Although SDSI/SPKI let us talk about *k*-out-of-*n* principals from a group, it does not let us talk about principals from different groups. There is no easy way, as in the following Binder rule

```
can(read, P, resource_r) :-
  vouched-for(P, D),
  vouched-for(P, R),
  senator(D, democrat),
  senator(R, republican).
```

to talk about access being vouched for by any one Democrat and any one Republican from the U.S. Senate.

7.3 PolicyMaker and KeyNote

Statements in PolicyMaker [4] and KeyNote [5] express conditions for granting access. This can be thought of as defining some abstract `can` predicate. PolicyMaker and KeyNote programs can state various conditions on the `can` predicate but cannot define additional lemma predicates, so they violate properties 2 and 3. For example, the `boss` example in Section 2 is difficult for PolicyMaker or KeyNote to encode. Binder lets us express the `boss` relation separately from `can`, while PolicyMaker and Key-

Note require us to collapse their definitions into its single `can` predicate.

PolicyMaker and KeyNote each construct a proof chain for a request, starting from the local policy, where each link of the chain can assert a filter (condition) on the request's parameters. One limitation of PolicyMaker and KeyNote is that this chain must be linear, while a Binder proof can be a directed acyclic graph (DAG). PolicyMaker and KeyNote also limit themselves to rules that state conditions on the request itself, and they cannot state conditions on other relations which may be lemmas to the request. Binder, in contrast, allows lemma predicates to be stated and composed.

Because PolicyMaker allows any programming language to be used to state policies, it fails property 5. Additionally, we cannot easily reason about PolicyMaker programs

7.4 SD3 and other logic-based security languages

Like Binder, SD3 is a security language based on datalog [13]. SD3 does not allow the transmission of rules in certificates, however; SD3 certificates can contain only facts. SD3 thus violates property 3.

D1LP [14] is also based on predicate calculus. It has a built-in treatment of "speaks-for" for delegation, but allows for the definition of other predicates that can be used in lemmas. D1LP does not allow the explicit construction of rules defining variants of delegation or for passing these rules in certificates; it therefore violates properties 2 and 3.

Appel and Felten have defined a security language based on a higher-order logic. Their system is more powerful than Binder but it has no decision procedure, and thus it violates property 4. Although undecidability is not a problem for a service if proofs come from the clients, where a given request might be more constrained and perhaps more decidable, we believe it would be impractical to require each request site to contain a significant amount of hand-crafted custom code to generate proofs.

7.5 DRM languages

Digital Rights Management languages (DRM languages) model consumers' access rights for digital media; XrML and ODRL are two examples [7, 11]. A DRM rule might give permission to play a movie two times, after paying $5. DRM rules can therefore talk about action (paying $5) as well as state (the number of plays remaining), while Binder cannot. Actions and state are difficult to discuss in a logic-based language, but we are currently investigating ways to extend Binder to handle these features of DRM languages.

Note that if multiple proofs are possible for an access request, but with different side-effects—for example, if different proofs draw on different accounts—then only the client may be in a situation to know which proof is preferable.

8. Experience with Binder

Most experience with Binder to date has involved writing small Binder programs, either to compare Binder with other security languages or using Binder as a language for expressing and comparing sample security policies. In particular, Binder has been used as a target for translating proposed security languages, in order to understand what statements Binder can express but these languages cannot, or vice versa. This work has included the prototyping of automated translators from these proposed languages to Binder, as well as the hand-translation of many examples.

Some features originally considered for Binder have been left out because they were not needed in our experience to date. This has resulted in a relatively simple language that is nevertheless as expressive as needed in our experience. Further experience is needed with the construction of large Binder programs to understand, for example, whether Binder's current limited mechanisms for the composition of rules are adequate or whether extending them could make large Binder programs easier to write or to understand.

Because Binder is close in form to Prolog, Binder programs can be translated into Prolog; we can simulate Binder's extra proof rules in a straightforward way. Binder programs have thereby been executed in an existing Prolog environment.

9. Future work

Is Binder strong enough? Binder may be too weak a language to model some real authorization problems; it might not be expressive enough to write certain security programs, or to write them well. For example, the wordiness caused by expressing all trust relations explicitly might complicate writing large security programs in Binder. Alternatively, Binder's current inability to talk about actions and state might become a problem. Further experience with writing large Binder programs will help us understand such possible problems. Strengthening Binder might involve strengthening the Binder logic, presumably by adding additional modal proof rules, such as direct support for predicates like "speaks-for."

It is also possible that Binder is already too strong a language. Although Binder provides powerful constructs, it may be too easy to misuse them and build a complex, incorrect security policy. It is possible that a simpler language might be easier to use and yet still be expressive enough in practice. Again, further experience will help us decide.

Although an open security language must be highly expressive, most of its uses will be application-specific and perhaps constrained. We might use Binder to define families of application-specific predicates that would be less powerful and less flexible, but easier for non-specialists to apply. While each application-specific family would be restricted in expressiveness, there would be no such restriction in the core language, and programs in these various families would interoperate via their ultimate definition in Binder. Again, more experience is needed to validate such an approach.

Acknowledgements

The author would like to thank Martín Abadi for his many helpful comments and insights on earlier drafts of this paper. The author would also like to thank Tony Hoare and the anonymous referees of the 2002 IEEE Symposium on Security and Privacy for their advice on improving the paper's presentation.

References

[1] M. Abadi and R. Needham. 1996. "Prudent engineering practices for cryptographic protocols," *IEEE Transactions on Software Engineering,* January 1996, pp. 6–15.

[2] M. Abadi, "On SDSI's linked local name spaces," *Proceedings of the 10th IEEE Computer Security Foundations Workshop,* Rockport, Mass., June 1997, pp. 98–108.

[3] A. Appel and E. Felten. "Proof-carrying authentication," *Proceedings of the 6th ACM Conference on Computer and Communications Security,* Singapore, November 1999, pp. 52–62.

[4] M. Blaze, J. Feigenbaum, and J. Lacy. "Decentralized trust management," *Proceedings of the 17th IEEE Symposium on Security and Privacy,* Oakland, Calif., May 1996, pp. 164–173.

[5] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. Keromyrtis. "The KeyNote Trust Management System Version 2," IETF RFC 2704, September 1999.

[6] Clocksin, W., and C. Mellish. *Programming in Prolog* (3rd ed.), Springer-Verlag, 1987.

[7] ContentGuard, Inc., *eXtensible rights Markup Language (XrML) 2.0 Specification*, available at http://www.xrml.org.

[8] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen, "SPKI certificate theory," IETF Network Working Group RFC 1693, September 1999.

[9] P. Gutmann, "X.509 style guide," available at http://www.cs.auckland.ac.nz/~pgut001/pubs/x509guide.txt, October 2000.

[10] J. Halpern, and R. van der Meyden, "A logic for SDSI's linked local name spaces," *Proceedings of the 12th IEEE Computer Security Foundations Workshop,* 1999, pp. 111–122.

[11] Iannella, R., editor, *Open Digital Rights Language (ODRL),* available at http://odrl.net.

[12] ITU-T Recommendation X.509, "The directory: public-key and attribute certificate frameworks," March 2000.

[13] T. Jim, "SD3: a trust management system with certified evaluation," *Proceedings of the 22nd IEEE Symposium on Security and Privacy,* Oakland, Calif., May 2001.

[14] N. Li, B. Grosof, and J. Feigenbaum, "A practically implementable and tractable delegation logic," *Proceedings of the 21st IEEE Symposium on Security and Privacy,* Oakland, Calif., May 2000, pp. 27–42.

[15] D. McDermott and J. Doyle, "Nonmonotonic logic I," *Artificial Intelligence*, 1980, pp. 41–72.

[16] R. Rivest and B. Lampson, "SDSI—a simple distributed security infrastructure," available at http://theory.lcs.mit.edu/~cis/ sdsi.html.

[17] J. Ullman, *Database and Knowledge-Base Systems,* volume 2, Computer Science Press, Rockville, Maryland, 1989.

Appendix A. EBNF grammar for Binder

```
<statement> ::= <clause>
<clause> ::= <head> [ ":-" <body> ] "."
<head> ::= <atom>
<body> ::= [ <atom> ("," <atom>)* ]
<atom> ::= [ <context> "says" ]
           <pred> [ "(" <args> ")" ]
<pred> ::= <constant>
<args> ::= <term> ("," <term>)*
<term> ::= <constant>
         | <variable>
<context> ::= <term>
<constant> ::= <lower> <idchar>*
             | """ <strchar>* """
<var> ::= <upper> <idchar>*
        | "_" <char*>
```

Here, `<upper>` is an upper-case letter; `<lower>` is a lower-case letter; `<char>` is any character; `<idchar>` is any character that can appear in an identifier; `<strchar>` is any character that can appear in a string.

The Binder grammar differs from a datalog grammar only in the optional quoting of atoms via `<context> says`. Quoting can appear only to depth 1; a quoted atom cannot be quoted again. Terms cannot be quoted at all. These restrictions are designed to interoperate with the rules for importing Binder statements, discussed below.

Appendix B. Semantics of Binder

The semantics of Binder are based on the semantics of datalog. We can transliterate a Binder program into datalog by moving the `says` quoting into an extra argument in every atom; `C says` *`pred(args)`* becomes *`pred`*`(C,` *`args`*`)`, while *`pred`*`(`*`args`*`)` becomes `pred(null,` *`args`*`)` where `null` is a new term that appears nowhere else in the program. After such a rewriting, we can adopt datalog's proof rules directly for Binder.

Under certain circumstances, a Binder statement from a context `C`—a fact, a rule, or a derivable atom—can be exported in a certificate signed by `C`, and imported into another context quoted by `C`. Below, we consider derivable atoms separately from facts and rules, as we extend datalog's standard proof rules with two additional proof rules for Binder (stated here informally).

B.1. Proof Rule 1

A certificate signed by `C` and containing a derivable atom that is not quoted with `says` (i.e., an atom of the form *`pred`*`(`*`args`*`)`) can be imported into any context, quoted with `C`. For example, the atom-bearing certificate

```
member(john_smith, bcl).
(signed: C)
```

can be imported as

```
C says member(john_smith, bcl).
```

An atom that is already quoted cannot be imported.

B.2 Proof Rule 2

A rule can be imported if the atom in its head is not quoted. A fact is equivalent to a rule with an empty body. When a rule in a certificate from context C is imported, its head will be quoted with `C`, and all unquoted atoms in its body will be quoted with `C`. For example, the rule-bearing certificate

```
member(X, bigco) :- member(X, bcl).
(signed: C)
```

can be imported as

```
C says member(X, bigco) :-
  C says member(X, bcl).
```

while the certificate

```
member(X, bigco) :-
  C' says member(X, bigco).
(signed: C)
```

can be imported as

```
C says member(X, bigco) :-
  C' says member(X, bigco).
```

Since an imported rule will have quoting in its head, an imported rule cannot be exported and imported again. Instead, the original certificate must be reused.

Appendix C. Time complexity of Binder

At any point in the execution of a Binder program at some context, the current rules and derivable atoms can be translated into datalog as described in Appendix B. Since datalog is decidable in polynomial time, there is a local polynomial-time decision procedure for Binder that ignores future communication.

While the restrictions on statement import in Binder may seem onerous, we suspect they may not be very significant in practice. We can imagine removing these restrictions, while at the same time generalizing Binder so that each atom can be quoted by zero or more terms, constant or variable, and terms can themselves be quoted by contexts to provide namespaces—as a generalization of SDSI—but so generalized a language would soon be no longer decidable. We suspect that there are lesser generalizations to Binder that retain a polynomial-time decision procedure, and we are currently exploring possible alternatives.