Declarative Reconfigurable Trust Management

Senior Project Final Report

William R. Marczak (wrm@seas.upenn.edu)
David Zook, Wenchao Zhou, Molham Aref
Faculty Advisor: Boon Thau Loo (boonloo@cis.upenn.edu)

Abstract

Recently, there have been a multitude of declarative trust management systems proposed, which have various tradeoffs depending on the security constructs they support (e.g. authentication, confidentiality, and delegation), and the assumed trust level and scale of the execution environment. Often, constructs are built in to the language and runtime, and cannot be easily added, removed, or reconfigured. Having more constructs makes a language more expressive, but also more complex. Often, there are benefits and drawbacks to configuring a construct in a certain way. Declaratively reconfigurable constructs enable easy modification, so designers of different systems can choose different tradeoffs, all with little or no change to the underlying framework. One example where reconfigurability is beneficial is in customizing the signature scheme used for authentication. In trust management languages such as Binder, asymmetric key signatures are used. However, to reduce the computational overhead associated with this scheme, it may be desired to trade some security for added performance by choosing a lightweight signature scheme such as message authentication codes or plaintext authentication. It may also be desired to utilize different schemes depending on the data being signed, or the user being communicated with. Another construct that benefits from reconfigurability is delegation, which may be extended by adding re-delegation control. While the addition of constraints on re-delegation certainly increases expressiveness, these constraints also mandate increased coordination between, and restrictions on, users, in ensuring that re-delegation control is respected. This project proposes the LBTrust system for reconfigurable trust management, built on top of LogicBlox, a commercial platform for enterprise applications. LogicBlox exposes a Datalog-like language, with several enhancements, such as loadable predicate libraries, schema constraints, static type-checking, and meta-programming.

1 Introduction

Trust management [2] is broadly defined as the process of assigning credentials (rights) to principals (users) to perform certain actions, and deciding whether credentials satisfy security policies. Trust management involves authentication—associating principals with statements and actions—delegation—delegating credentials among principals to establish chains of trust—as well as other concepts such as secrecy and integrity. A trust management system provides security constructs that encode a subset of these concepts, a language that supports these constructs, for expressing policies and credentials, and a mechanism to determine whether credentials satisfy policies.

Over the years, logical ideas and tools have been used to explain and improve trust management, particularly to implement access control in a multi-user distributed environment. Several declarative logic-based languages (e.g. Binder [4], Cassandra [1], DiLP [6], have been proposed to ease the process of expressing, analyzing, and encoding security policies. These proposals have different tradeoffs in expressiveness and complexity, depending on the security constructs that they support, as well as the trust level and distribution of their assumed execution environments. Binder, one of the simplest languages, supports the logical says operator for authentication, where principals that assert facts must first validate their identity in a secure fashion. Other trust management systems have explored additional security constructs for secrecy and encrypted facts, ensuring that only authorized principals can interpret facts in distributed settings.

In order to differentiate principals based on their capabilities, one can further incorporate the
notions of *speaks-for* and *restricted delegation* into the language, which allow principals to delegate the responsibility of selected policy decisions to other principals.

All in all, there is an inherent tradeoff between expressiveness and complexity across these spectrum of languages, and while each language may be intriguing in isolation, one would not want to combine them all indiscriminately.

This project proposes *LBTrust*, a unified declarative system for reconfigurable trust management, where various logical security constructs can be customized and composed using a variant of the Datalog language. In this report, we present an implementation of *LBTrust* using *LogicBlox*, an emerging commercial Datalog-based platform for enterprise software systems. The LogicBlox language provides several enhancements to Datalog, including *constraints* and *meta-programming*. LogicBlox’s meta-programming is based on a *meta-model* similar to the recently proposed Evita Raced, which features a bootstrapped meta-circular compiler implemented in Datalog. Unlike Evita Raced, LogicBlox also supports applying programmer-defined constraints to the meta-model – *meta-constraints* – which act to restrict the allowed programs. *LBTrust* utilizes LogicBlox’s meta-programming and meta-constraints to enable customizable cryptographic, partitioning and distribution strategies based on the deployed environment.

In *LBTrust*, it is possible to express a wide variety of security constructs for authentication, confidentiality, integrity, speaks-for, and restricted delegation (used in D1LP). Based on these primitives, one can support several different trust management systems. In this report, we will show use cases involving the Binder trust management system, and the *Secure Network Datalog* declarative networking language.

Also, because it is a unified declarative platform, *LBTrust* provides a basis for comparison across different trust management systems, and potentially provides avenues for better analyzing security properties across these various languages.

## 2 Background and Related Work

### 2.1 Datalog

First, we review Datalog, following the conventions in Ramakrishnan and Ullman’s survey. A Datalog program consists of a set of declarative *rules*. Each rule has the form $p \leftarrow q_1, q_2, \ldots, q_n$, which can be read informally as “$q_1$ and $q_2$ and … and $q_n$ implies $p$”. Here, $p$ is referred to as the *head* of the rule, and $q_1, q_2, \ldots, q_n$ is a list of *literals* that constitutes the *body* of the rule. A literal is a possibly negated atom. An atom is a *predicate* applied to a list of *terms*, each of which is either a *constant* or a *variable*. The names of predicates, function symbols, and constants begin with a lowercase letter, while variable names begin with an uppercase letter. Negation may not occur in the head of a rule, and in the body it must be *safe* – every variable occurring in a negated literal must also occur somewhere in a non-negated literal. Also, for readability, *solitary* variables – those which occur just once in a rule – are often replaced with an underscore (_).

Each predicate occurring in the head of a rule is called *intensional*, while all the other predicates are called *extensional*. A Datalog program takes as input an assignment of values of the extensional predicates and derives a minimal model of the intensional predicates consistent with the logical meaning of the rules.

LogicBlox (and most Datalog implementations) provide built-in functions for equality, arithmetic, and aggregation (totaling and counting), as well as built-in predicates for common types (numbers and strings). Also, it easily can be shown that an arbitrary nesting of negation, conjunction, and disjunction may be used in the body of a rule. Such a rule may be translated into strict Datalog rules by (1) translating the body into Disjunctive Normal Form (DNF), and (2) splitting the original rule into a separate rule for each resulting alternative, duplicating the original head. We use a left-arrow ($\leftarrow$) for logical implication, a comma (,) for conjunction, a semicolon (;) for disjunction, a bang (!) for negation, and parentheses for grouping.
2.2 Logic-based Trust Management

To illustrate logic-based trust management languages, we provide an example from the Binder [4] language. A Binder program is a set of Datalog-style logical rules. Binder extends Datalog by providing a distinguished operator called 

\texttt{says} – used for expressing authentication – and the notion of a \texttt{context} – a component in a distributed system. A simple policy in Binder is:

\begin{verbatim}
  at alice:
  b1: access(P,O,read) <- good(P).
  b2: access(P,O,read) <- bob says access(P,O,read).
\end{verbatim}

The above rules \texttt{b1} and \texttt{b2} can be read as “any principal \texttt{P} may access any object \texttt{O} in \texttt{read} mode if \texttt{P} is good or if \texttt{bob} says that \texttt{P} may do so”. The logical meaning of \texttt{says} is that if a principal \texttt{p} supports a statement \texttt{s}, then we assert “\texttt{p says s}.” Note that the \texttt{says} operator abstracts from the details of authentication.

A principal in Binder refers to a component in a distributed system. Each principal has its own local \texttt{context} where its rules reside. Binder assumes an \texttt{untrusted} network, where different components can serve different roles, running distinct sets of rules. Because of the lack of trust among nodes, a component does not have control over rule execution at other nodes. Instead, Binder allows separate programs to interoperate correctly and securely via the export and import of rules and derived tuples across contexts. For example, rule \texttt{b2} can be a local rule that is executing in the context of principal \texttt{alice}, which imports derived \texttt{access} tuples from the principal \texttt{bob} into its local context via \texttt{bob says access(p,o,read)} in its rule body.

Binder specifies an asymmetric key signature scheme, such as RSA, for the \texttt{says} construct. In a hostile world, \texttt{says} may require this, but in a more benign world, one may wish to trade some security for efficiency, and configure \texttt{says} to use some lightweight signature scheme such as HMAC, or simply to append plaintext principal headers to messages. Somewhere in between, the use of cryptographic signatures may be applied only to certain important messages, or when communicating with specific principals. Binder does not provide any leverage in deciding how this tradeoff should be made.

In addition to constructs for authentication, declarative trust management systems often feature security constructs for \texttt{integrity}, \texttt{secrecy}, and \texttt{delegation}. The D1LP [6] language further supports constructs that implement distributed vote-based agreement, where a fact in the rule head is derived only when \texttt{k-out-of-n principals} in a rule body predicate derive a similar fact concurrently.

3 Architecture

\texttt{LBTrust} is implemented using \texttt{LogicBlox}, a commercial platform for building enterprise-scale corporate planning and pricing applications, which feature analyses requiring aggregation across very large data sets, combined with simulation and modeling techniques.

LogicBlox contains a Datalog-based logic programming language enhanced with a variety of features, including functional dependencies, aggregation functions, schema constraints, static type-checking, tuple-generation, temporal logic support, predicate partitioning, distributed computation, and meta-programming. LogicBlox further allows application-defined libraries of custom predicates to be imported, such as the cryptographic functions required for implementing certain security constructs. Also essential for these constructs is LogicBlox’s support for applying schema constraints to the meta-model, which we call \texttt{meta-constraints}. To set the stage for presenting \texttt{LBTrust}'s implementation of various security constructs, this section explains LogicBlox’s facilities for supporting schema constraints, meta-programming, and distributed computation.

3.1 Execution Environment

LogicBlox utilizes a bottom-up semi-naïve fixpoint [9] execution model for executing Datalog programs. LogicBlox provides an interface for submitting a program for compilation and execution within a \texttt{workspace}. A \texttt{workspace} in LogicBlox is essentially a database instance which contains
a set of predicate definitions and a set of active rules. Within a designated workspace, the LogicBlox API allows an application to query and modify the data defined by the workspace, including adding/removing facts and rules. When predicate data is modified, the active rules are incrementally recomputed.

3.2 Constraints

Unlike a rule that calculates new values for a predicate, a schema constraint restricts a predicate’s allowed values. LogicBlox adds schema constraints to Datalog by means of the special predicate fail(). If any rule defines fail() to be true, then the evaluation of the Datalog program fails by terminating with an error, and all changes to the workspace in the current transaction are rolled back.

For example, a schema constraint for the Binder program given above might require that any value occurring in the first argument of the access predicate also occur in the principal predicate. This constraint can be expressed as a rule:

\[
\text{fail() } \leftarrow \text{access}(P,O,M), \neg \text{principal}(P).
\]

This rule defines fail() to be true if, for any assignment of values to the variables P, O, and M, the atom access(P,O,M) is true but principal(P) is false.

Constraints expressed using fail() can often be unintuitive. So as a notational convenience LogicBlox supports a logically equivalent positive form for constraints, indicated with a right arrow (\(\rightarrow\)). If \(F_1\) and \(F_2\) are arbitrary nestings of conjunction, disjunction and negation, then the logical meaning of \(F_1 \rightarrow F_2\) is \(\text{fail()} \leftarrow F_1, \neg(F_2)\). For example, the positive form of the constraint given above is:

\[
\text{access}(P,O,M) \rightarrow \text{principal}(P).
\]

Typically every argument of access would be constrained:

\[
\text{access}(P,O,M) \rightarrow \text{principal}(P), \text{object}(O), \text{mode}(M).
\]

Informally, this may be read as “for any values of P, O and M, whenever access(P,O,M), then require principal(P) and object(O) and mode(M).” In fact, in LogicBlox, a type is considered to be a unary predicate (representing a set of values). Hence, in LogicBlox, this kind of schema constraint acts as a type declaration. The use of types and type-checking (statically, and dynamically when rules are added to workspaces) ensures that only type-safe LogicBlox programs are executed.

3.3 Meta-Programming

LogicBlox features a meta-model—a collection of predicates that represent the currently executing program. The type declarations for these predicates are shown in Figure 1. Furthermore, each LogicBlox workspace contains an active predicate that stores the rule identifiers of active rules within the workspace.

The meta-model is significant because programmer-defined rules may refer to the meta-model. For example, an active rule may perform reflection (i.e. inspecting the program’s structure) by referring to meta-model predicates in its body. Or, a rule may perform code generation (adding or rewriting existing rules) by referring to the meta-model in its head. If the evaluation of a rule derives a new fact in the active predicate, then the associated meta-model facts are turned into a new rule which is added to the program and evaluated. Note that when one inserts a rule identifier into the active predicate, one must also specify the associated meta-model facts (e.g. head, body, atom, etc.) that determine the contents of the rule.

A special case of reflection is a schema constraint that refers to meta-model predicates (a meta-constraint). While meta-constraints are usefully generally for imposing integrity constraints similar to those in databases, they are particularly useful in the context of LBTrust for expressing security restrictions. To illustrate, assume we wanted to require that a principal may only read predicates
that he or she is authorized to access. Datalog (without constraints) provides no way to enforce this requirement, because there is no way to prohibit an attempted access.

To support this restriction in LBTrust, we first define an owner predicate that associates a rule with the principal that added that rule, and an access predicate that represents access rights to a predicate. Note that we leverage predicate, a meta-model predicate that contains a unique entry for each predicate defined in the workspace.

\[
\text{owner}(U, R, P) \rightarrow \text{rule}(R), \text{principal}(P).
\]

\[
\text{access}(U, P, M) \rightarrow \text{principal}(U), \text{predicate}(P), \text{mode}(M).
\]

With our schema defined, we then apply the following meta-constraint, which provides the desired prohibition. It says that for any principal \( U \) who owns a rule with predicate \( P \) in the body, there must be a fact in the access predicate granting \( U \) the right to read \( P \).

\[
\text{owner}(U, \{ | A \leftarrow \neg P(T*), A* | \}) \rightarrow \text{access}(U, P, \text{read}).
\]

The above example illustrates the use of the quoted code term – a rule or atom surrounded by the code-quotes: \([ \] \) and \([ | \] \). Inside the code-quotes is a code pattern that matches one or more rules. The star (*) represents the Kleene star – a repetition of the pattern preceding it (zero or more times). The capital letters in the pattern are meta-variables – variables that represent pieces of code. The types of the meta-variables are determined by their position in the pattern. In this example, \( A \) takes the place of an atom, \( T \) takes the place of a term, and \( P \) takes the place of a predicate.

The LogicBlox compiler translates the code inside the code-quotes into a conjunction of atoms on the meta-model representing the quoted code. For example, the above meta-constraint translates into:

\[
\text{owner}(U, R1), \text{rule}(R1), \text{body}(R1, A1), \\
\text{atom}(A1), \text{functor}(A1, P) \rightarrow \text{access}(U, P, \text{read}).
\]

The variables \( R1 \) and \( A1 \) are freshly generated by the translation. Note that the meta-variable \( P \) occurs outside the quoted code. The tilde (\( \sim \)) indicates that its value is to be unquoted and bound to the \( P \) appearing outside the pattern.

### 3.4 Partitioning and Distribution

Partitioning is a mechanism for logically separating facts based on their attributes. In trust management systems, data is partitioned by principal, with each partition typically called a principal’s
A context stores all facts local to a principal, and communication between contexts represents communication between principals.

In a distributed setting, principals may reside on different nodes, and the execution of security policies may result in an exchange of rules (similar to Binder’s transfer of rules across contexts). In LogicBlox, logical partitioning and distribution are separated, hence providing location transparency, where a multi-principal security policy can be distributed in a customized fashion based on the deployed execution environment (e.g. single vs multiple principals per physical host). This separation is accomplished by allowing the user to define a custom placement policy for a partitioned predicate.

In order to support partitioning and distribution, predicates that contain facts spread across multiple principals specify a special built-in node type in their keyspace. In general, consider a predicate \( p \) with \( n \) arguments where \( t_1(X_1) \) denotes the type of \( X_1 \):

\[
p(X_1, \ldots, X_n) \rightarrow t_1(X_1), \ldots, t_n(X_n).
\]

Partitioning \( p \) entails creating a new predicate \( p' \) with a node argument:

\[
p'(N, X_1, \ldots, X_n) \rightarrow \text{node}(N), t_1(X_1), \ldots, t_n(X_n).
\]

Facts asserted into \( p' \) will automatically be sent to the node specified in the first predicate argument. One can distribute data by defining a placement policy to populate \( p' \) with values from \( p \). For example, if one has a mapping, \( x_1\text{node} \) from one of the key arguments, \( X_1 \) to the set of nodes, the following placement policy will provide the desired distribution of facts in \( p \):

\[
p'(N, X_1, \ldots, X_n) \leftarrow p(X_1,\ldots,X_n), x_1\text{node}(X_1,N).
\]

This rule takes each value of \( X_1 \), looks up its associated network node \( N \), and then places the corresponding subset of \( p \) into the \( p' \) predicate on \( N \).

## 4 Security Primitives in LBTrust

In this section, we demonstrate how various security primitives can be customized and supported by LBTrust. This is by no means intended to be an exhaustive coverage of the possibilities. Our main goal here is to illustrate the key language features of LBTrust, and highlight the flexibility and compactness of LBTrust in supporting various security primitives. We will build upon these primitives in our next section when we present case studies of languages enabled by LBTrust.

### 4.1 Authentication

Authentication is a central component of security, where the identity of a principal is established and verified. Authentication is essential for authorization, where an authenticated principal is granted access to perform actions on shared resources. Practically all logic-based trust management languages provide some language support for expressing authentication, typically via a distinguished operator called says, which associates a principal with a statement (described in Section 2). In each case, the semantics of says are hardwired into the system as an add-on to the logic programming language. In LBTrust, however, the says concept is configured in the same language as the policy, using features not specifically designed to support security concerns – except for cryptographic primitives used to implement various authentication schemes.

The simplest way to associate a principal with every fact in a predicate \( P(T^*) \) is to add an extra argument representing the principal who said the fact: \( P(U, T^*) \). However, this necessitates changing the schema of all of the predicates. In LBTrust, we represent says as a meta-predicate, says\((U_1, U_2, R)\), which associates a Datalog rule \((R)\) with both the source principal who said the rule \((U_1)\), and the destination principal to whom the rule is said \((U_2)\). Note that while communication occurs in the form of rules, we can also communicate facts (rules with an empty body). We define the says predicate below, by showing its type constraint:

\[
says0: \text{says}(U_1,U_2,R) \rightarrow \text{prin}(U_1), \text{prin}(U_2), \text{rule}(R).
\]
Using the \textit{says} predicate, authorization may be implemented with some simple meta-constraints. The following constraints restrict read and write access to predicates respectively:

\begin{align*}
says(U,me \mid A \leftarrow \neg P(T*), A* \mid) &\rightarrow \text{mayRead}(U,P). \\
says(U,me \mid \neg P(T*) \leftarrow A* \mid) &\rightarrow \text{mayWrite}(U,P). 
\end{align*}

4.1.1 Authenticated Communication

To enable communication between principals, we introduce the \textit{export} predicate as well as additional meta-rules and meta-constraints. The following rules implement rule export using the RSA authentication scheme (\texttt{rsasign} and \texttt{rsaverify}).

\begin{align*}
\text{exp0}: \text{export}(N1,U2,R,S) &\rightarrow \text{node}(N1), \text{prin}(U2), \text{rule}(R), \text{string}(S). \\
\text{exp1}: \text{export}(N2,me,R,S) &\leftarrow \text{says}(me,U2,R), \text{rsasign}(R,S,K), \text{rsaprivkey}(me,K), \\
&\quad \text{prinnode}(U2,N2).
\end{align*}

Consider a local principal \textit{me} that wishes to export rule \textit{R} to another principal \textit{U2} via the \textit{says} predicate. Rule \texttt{exp0} declares the type definitions of the \textit{export} predicate that will be used for exporting the rule \textit{R} with its signature to the destination principal \textit{U2}. \texttt{export} has a placement policy that assigns the location of each partition to match the location of the corresponding principal—by looking up a principal’s associated node in the \texttt{prinnode} predicate.

Rule \texttt{exp1} calculates the appropriate RSA signature \textit{S} using the private key of the local principal, and copies the rule into the destination principal's partition of the \textit{export} predicate. The following rules would then run at the destination principal (\textit{U2}), to import received rules. Note that in this case, \textit{me} refers to principal \textit{U2}.

\begin{align*}
\text{exp2}: \text{says}(U,me,R) &\leftarrow \text{export}(N,U,R,S), \text{prinnode}(me,N). \\
\text{exp3}: \text{says}(U,me,R) &\rightarrow \text{export}(N,U,R,S), \text{rsapubkey}(U,K), \text{rsaverify}(R,S,K), \\
&\quad \text{prinnode}(me,N).
\end{align*}

Rule \texttt{exp2} copies the received rule from the \textit{export} predicate into the local \textit{says} predicate. Finally, \texttt{exp3} verifies the signature of the new rule using the source principal’s public key.

4.1.2 Alternative Authentication Schemes

Because the signature generation and verification has been defined in a series of Datalog rules, it is easy to replace the RSA scheme above with an alternate scheme. To illustrate, we demonstrate signing each message with a \textit{keyed-hash message authentication code} (HMAC), typically a cryptographic hash of the message data and a secret key shared between the two communicating principals. The choice of an alternative signature generation scheme is often a tradeoff between security and performance. For example, HMAC is computationally less expensive but requires the use of shared symmetric keys among principals that wish to communicate. Also, the use of a symmetric key to generate a rule signature implies that the signature will only be verifiable at principals that have the same key as the signer. The following rules implement an HMAC signature scheme.

\begin{align*}
\text{exp1}': \text{export}(N2,me,R,S) &\leftarrow \text{says}(me,U2,R), \text{hmacsign}(R,K,S), \text{sharedsecret}(me,U2,K), \\
&\quad \text{prinnode}(U2,N2). \\
\text{exp3}': \text{says}(U,me,R) &\rightarrow \text{export}(N,U,R,S), \text{sharedsecret}(me,U,K), \text{hmacverify}(R,S,K), \\
&\quad \text{prinnode}(me,N).
\end{align*}

Interestingly, only two rules (\texttt{exp1'} and \texttt{exp3'}) need to be modified, while the trust policies that utilize the \textit{says} predicate remain unchanged, demonstrating the ease with which new authentication schemes can be enabled by \texttt{LBTrust}.

4.1.3 Confidentiality and Integrity

\texttt{LBTrust} can support confidentiality, ensuring rules cannot be interpreted by unauthorized principals in a distributed setting, and integrity, ensuring data is not corrupted in transit. This requires the addition of built-in predicates representing various encryption and integrity schemes such as checksums and cryptographic hashes.
4.2 Delegation

Often-times in security, it is useful to establish a chain of trust among different principals. This is particularly useful for performing delegation, where different principals may choose to assign capabilities to other principals, either for performance, accessibility, or security reasons. For example, a principal may delegate the authority to associate principals with public keys to a certificate authority. Or a credit card issuer may wish to delegate authority to a credit rating agency to associate a credit score with an individual.

An early version of delegation is the speaks-for construct. Adopting the definition based on Lampson’s security survey paper [5], speaks-for works as follows. Consider a statement of the form “principal U1 speaks for principal U2.” The logical meaning behind “speaks for” is that if U1 says something, then U2 says it too. So for example, alice can say that bob speaks for her by activating the following meta-rule, which activates any rule R said by bob:

\[
\text{sf0: active(R) } \leftarrow \text{says(bob,me,R).}
\]

A speaks-for rule is a special case of delegation where a principal delegates all authority to another principal. In practice, it is useful to restrict this delegation to a specific predicate. To ease the specification of complex delegation policies, LBTrust defines a delegates predicate whose type declaration is shown in del0. In the example below, delegates(U1,U2,P) denotes that U1 delegates the responsibility of deriving predicate P to U2 – essentially expressing the speaks-for construct where U2 speaks for U1 with respect to P.

\[
\text{del0: delegates(U1,U2,P) } \rightarrow \text{ prin(U1), prin(U2), predicate(P).}
\]

The del1 expresses that whenever a delegation fact is added, the appropriate speak-for rule is automatically generated.

4.2.1 Delegation Depth and Width

Sometimes it is useful to restrict delegation authority [6]. For instance, we may restrict one or both of the depth – the maximum permitted length of the delegation chain – and the width – the set of principals allowed to be part of the chain.

The following meta-rules declare and enforce a delDepth predicate. The inferredDelDepth predicate takes the originally specified delegation depths and infers new depth restrictions. The base case is when a principal U1 delegates to U2 with depth limitation N=0. In this case, as expressed by meta-constraint dd4, any delegation by U2 conflicts with the limitation. The recursive case, as expressed by the rule defining inferredDelDepth, is when U1 delegates to U2 with a depth limitation of N > 0. In this case, if U2 delegates to some other principal U3, then a new limit of N-1 is inferred between U2 and U3. Similar meta-rules can be formulated to enforce delegation width restrictions.

\[
\text{dd0: delDepth(U1,U2,P,N) } \rightarrow \text{ prin(U1), prin(U2), predicate(P), int[64](N).}
\]

\[
\text{dd1: inferredDelDepth(U1,U2,P,N) } \rightarrow \text{ prin(U1), prin(U2), predicate(P), int[64](N).}
\]

\[
\text{dd2: inferredDelDepth(me,U,P,N) } \leftarrow \text{ delDepth(me,U,P,N).}
\]

\[
\text{dd3: says(me,U,[| inferredDelDepth(me,"U","P","N-1). |]) } \leftarrow \text{ inferredDelDepth(me,U,P,N),}
\]

\[
\text{delegates(me,U,P), N>0.}
\]

\[
\text{dd4: inferredDelDepth(_,me,P,0) } \rightarrow \text{ !delegates(me,_,P).}
\]

An interesting case arises if a non-conforming delegation exists before a delegation depth restriction is added. The rule dd3 will propagate an inferred delegation depth of 0 to the principal with the non-conforming delegation, causing a violation of the dd4 constraint. However, none of the principals in the delegation chain up to that point will be aware of the violation.

4.2.2 Delegation Thresholds

Another delegation variant is the use of threshold structures. An unweighted threshold structure will authorize some operation if any k out of n principals concur. For example, a bank may consider
a customer’s credit okay if at least three credit bureaus do. This is easily expressed in LBTrust using
the count aggregation:

\[
\begin{align*}
\text{wd0: } & \text{creditOK}(C) \rightarrow \text{customer}(C). \\
\text{wd1: } & \text{creditOK}(C) \leftarrow \text{creditOKCount}(C,N), \text{ } N \geq 3. \\
\text{wd2: } & \text{creditOKCount}(C,N) \leftarrow \text{agg}<N = \text{count}(U)>(\text{prigroup}(U,\text{creditBureau}), \\
& \text{says}(U,\text{me}, [\mid \text{creditOK}(\sim C).]) ).
\end{align*}
\]

It is also straightforward to generalize these rules to handle more sophisticated threshold structures such as weighted delegation, where different credit bureaus have different reliability factors assigned to them. Rule \text{wd2} above would be modified to use the total aggregation.

5 Case Studies

In this section, we focus on how LBTrust can leverage the basic security constructs presented in the previous section to implement trust management languages. We focus on two case studies: Binder and the Secure Network Datalog (SeNDlog) language used in declarative networking.

5.1 Binder

As described in Section 2.2, Binder is a logic-based trust management system that extends Datalog with the says construct and the notion of communication across contexts. Each component, or principal, in the distributed system has a local Binder context. Binder contexts correspond to LogicBlox workspaces described in Section 3. To authenticate facts asserted by principals, Binder uses certificates signed with the private key of the sending principal. Certificates are imported by prefixing the says operator with a public key representing the context to import from. In our implementation of Binder, we use the says predicate defined in Section 4.

To illustrate, the LBTrust equivalent to the Binder rule b2 presented in Section 2.2 is the following:

\[
\begin{align*}
\text{b2': } & \text{access}(P,O,\text{read}) \leftarrow \text{says}(\text{bob},\text{me},[\mid \text{access}(\sim P,\sim O,\text{read}) ].)
\end{align*}
\]

Top-down to Bottom-up Rewrite: Most practical access control languages, including Binder, utilize a top-down evaluation strategy. Specific requests are made as goals, which are then resolved against the security policies, hence minimizing the disclosure of sensitive information. Top-down evaluation means that Binder access control rules may “pull” rules in from other nodes, whereas LogicBlox’s bottom-up evaluation strategy implies that rules are only “pushed” to other nodes. One possible approach that we are exploring to emulate top-down evaluation in LBTrust is converting a “pull” request in the body of a rule into two “pushes”. The following meta-rules express this automatic conversion:

\[
\begin{align*}
\text{pull0: } & \text{says}(\text{me},X,[\mid \text{request}(\sim R).]) \leftarrow \text{active}([\mid A \leftarrow \text{says}(\sim X,\text{me},\sim R), A*. ])), X=\text{me}. \\
\text{pull1: } & \text{says}(\text{me},X,R) \leftarrow \text{says}(X,\text{me},[\mid \text{request}(\sim R).])
\end{align*}
\]

Rule pull0 matches any rule \text{R} that has says in the body, and pushes a request to \text{X}. Rule pull1 responds to a request by pushing back the desired data.

5.2 Secure Network Datalog

SeNDlog [10] is a unified declarative language for network specifications and security policies, which combines the Network Datalog language used in declarative networking with Binder. Similar to Binder, SeNDlog allows different principals or contexts to communicate via import and export of tuples. To differentiate from local predicates, an import predicate from a principal \text{N} is quoted using “\text{N says}”, whereas an export predicate of the form “\text{p@X}” in a rule head indicates the predicate \text{p} is exported to the principal \text{X} from the context where it is derived. An example of SeNDlog is shown below, in rules s1-s2, which compute all pairs of reachable nodes in a network:
at S:

\begin{align*}
s1: \text{reachable}(S,D) :& \leftarrow \text{neighbor}(S,D) . \\
s2: \text{reachable}(Z,D) & @Z : \leftarrow \text{neighbor}(S,Z), \ W \ says \ \text{reachable}(S,D) .
\end{align*}

Rule s1-s2 are executed in the context of node S. Rule s1 takes neighbor tuples as input to compute one-hop reachable tuples. Rule s2 specifies a distributed transitive closure computation, expressing that “if Z is a neighbor of S, and S can reach D, then Z can also reach D.” Unlike an ordinary transitive closure computation, the above Sendlog program is authenticated (via the use of “says”) and distributed via the use of import and export predicates. By modifying this simple example, one can easily construct more complex secure networking protocols, such as an authenticated path-vector protocol.

Given the says predicate described in Section 4, the LBTrust equivalent of the above Sendlog rules is as follows:

\begin{align*}
lc1: \text{neighbor}(S,D) & \rightarrow \text{prin}(S), \ \text{prin}(D) . \\
lc2: \text{reachable}(S,D) & \rightarrow \text{prin}(S), \ \text{prin}(D) . \\
ls1: \text{reachable}(me,D) & \leftarrow \text{neighbor}(me,D) . \\
ls2: \text{says}(me,Z,[ | \text{reachable}(\text{~Z,~D}). | ] \leftarrow \text{neighbor}(me,Z), \\
& \text{says}(W,me,[ | \text{reachable}(me,D). | ])
\end{align*}

Using LogicBlox’s support for distribution described in Section 3.4, one can customize the locations of principals by customizing the export predicate, enabling the support of various distribution plans. Note that distribution is not required for the neighbor and reachable tables since they are only used locally at each node (and hence no partitioning is strictly necessary).

6 Conclusion

In this project, we developed LBTrust, a unified declarative system for reconfigurable trust management, where various security constructs can be customized and composed in a declarative fashion. We applied meta-programming and constraints to enforce access control policies, and build many common security constructs. We demonstrated how different existing trust management languages could be expressed in our system, and exhibited use cases to illustrate how these languages could be composed in LBTrust.

References


The Cassandra paper develops a trust management system using tunable expressiveness, which allows policies to be specified with a certain constraint domain. Choosing a more expressive constraint domain allows more complicated policies. Their notion of reconfigurability is different from ours, as we strive to enable customizability on the security construct level, not in restricting the types of atoms in a rule body.


This paper introduced the concept of trust management, and the first trust management system—PolicyMaker. The paper explores existing systems, and shows the benefits of having a unified system, such as enabling separation between application-defined policy, and a generic system for evaluating credentials and policy statements. While this paper does not represent the most current work in trust management, nor
is the policy specification language declarative, the core concepts of the system are still relevant in the field.


The Evita-Raced paper illustrates how a compiler for a Datalog-like language used for declarative networking, Overlog, can be written in the same language. The paper introduces the concept of a metacompiler catalog, which is a set of predicates that represent an Overlog program, similar to the LogicBlox meta-model. The paper provided some insight on how to implement the meta-execution algorithm.


The paper introduces one of the first logic-based trust management languages, using a Datalog-like language to express security policies in a novel fashion. While the language is designed to be more expressive than security languages existing at the time of publication, DeTreville is careful to not haphazardly expand expressiveness, acknowledging that if a language is too powerful, people may misuse it. In addition to providing many examples of statements that can be expressed, the author illustrates the limits of Binder through use cases such as expressing a DRM policy. The paper has heavily influenced the design of LBTrust.


This paper provides a comprehensive review of security constructs in the context of operating systems and networks. An interpretation of many security constructs is provided, along with associated axioms. The constructs are motivated through numerous examples. While the paper is from 1992, its contents is still relevant. The paper helped guide my thinking about security constructs such as speaks-for.


This paper presents D1LP, an extension of Datalog that supports advanced security constructs, such as restricted delegation, and threshold structures. The paper argues that existing systems do not provide enough flexibility, particularly in their support of delegation, and provides several use cases to suggest that a more robust delegation construct is needed. The paper briefly treats non-monotonic constructs, explaining how they might be supported, but D1LP itself is purely monotonic. This paper helped me in evaluating the expressiveness of LBTrust.


This paper was published in CIDR, a biennial top-tier systems oriented conference, and describes the beginning of the implementation of my project. It provides an overview of some current declarative trust management systems, showing how these systems’ constructs can be represented in a unified system through two case studies. The paper also describes how construct specification, and distribution policy, are separated from the application, allowing one to change the implementation of constructs, or the location of principals, without rewriting their application. This final report borrows largely from the paper.
This survey paper from 1993 looks at a wide range of deductive database systems. Of particular interest is its review of Datalog. It provides a syntactic description, as well as a description of evaluation algorithms, including semi-naive evaluation, and optimization techniques such as magic sets.

This paper introduces the Secure Network Datalog (SeNDLog) language, which is a combination of NDLog, a language for declarative networking, with Binder-like syntax. By expressing security policies and credentials in the same language as the networking protocol, developers have one fewer language to learn, and gain more precise control over the interaction between security policy and network protocols. The paper presents an evaluation algorithm for the language, called APSN, which extends the traditional Pipelined Semi-Naive algorithm with mechanisms for verifying digital signatures. In essence, my project is the next logical step of the work in this paper, allowing security constructs to be expressed in the same language as the security policy, and application.