SecureBlox: Customizable Secure Distributed Data Processing

William R. Marczak
University of California - Berkeley

Shan Shan Huang
LogicBox, Inc.

Martin Bravenboer
LogicBox, Inc.

Micah Sherr
Georgetown University

Boon Thau Loo
University of Pennsylvania, boonloo@cis.upenn.edu

See next page for additional authors

Abstract

We present SecureBlox, a declarative system that unifies a distributed query processor with a security policy framework. SecureBlox decouples security concerns from system specification, allowing easy reconfiguration of a system's security properties to suit a given execution environment. Our implementation of SecureBlox is a series of extensions to LogicBlox, an emerging commercial Datalog-based platform for enterprise software systems. SecureBlox enhances LogicBlox to enable distribution and static meta-programmability, and makes novel use of existing LogicBlox features such as integrity constraints. SecureBlox allows meta-programmability via BloxGenerics–a language extension for compile-time code generation based on the security requirements and trust policies of the deployed environment. We present and evaluate detailed use-cases in which SecureBlox enables diverse applications, including an authenticated declarative routing protocol with encrypted advertisements and an authenticated and encrypted parallel hash join operation. Our results demonstrate SecureBlox's abilities to specify and implement a wide range of different security constructs for distributed systems as well as to enable tradeoffs between performance and security.

SecureBlox: Customizable Secure Distributed Data Processing

William R. Marczak[*] Shan Shan Huang[†] Martin Bravenboer[†]
Micah Sherr[‡] Boon Thau Loo[‡] Molham Aref[†]

[*]*University of California, Berkeley:* wrm@berkeley.edu
[†]*LogicBlox. Inc:* {ssh, martin.bravenboer, molham.aref}@logicblox.com
[‡]*University of Pennsylvania:* {msherr, boonloo}@cis.upenn.edu

ABSTRACT

We present *SecureBlox*, a declarative system that unifies a distributed query processor with a security policy framework. SecureBlox decouples security concerns from system specification, allowing easy reconfiguration of a system's security properties to suit a given execution environment. Our implementation of SecureBlox is a series of extensions to *LogicBlox*, an emerging commercial Datalog-based platform for enterprise software systems. SecureBlox enhances LogicBlox to enable distribution and static meta-programmability, and makes novel use of existing LogicBlox features such as integrity constraints. SecureBlox allows meta-programmability via *BloxGenerics*–a language extension for compile-time code generation based on the security requirements and trust policies of the deployed environment. We present and evaluate detailed use-cases in which SecureBlox enables diverse applications, including an authenticated declarative routing protocol with encrypted advertisements and an authenticated and encrypted parallel hash join operation. Our results demonstrate SecureBlox's abilities to specify and implement a wide range of different security constructs for distributed systems as well as to enable tradeoffs between performance and security.

Categories and Subject Descriptors

H.2.7 [**Database Management**]: Database Administration—*Security, integrity, and protection*; H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Design, Languages, Security

Keywords

Datalog, Secure data management, Distributed query processing

1. INTRODUCTION

In recent years, there has been a proliferation of large-scale networked information systems for a variety of application domains including network monitoring infrastructures, publish-subscribe systems, cloud computing, content distribution networks, and network routing. These systems typically involve the continuous processing of data in a distributed setting.

Recent work has shown that such systems can be specified as queries in high-level declarative languages based on Datalog, often in orders of magnitude less code than their imperative implementations. For example, one can express routing [23] and concurrency [5] protocols, overlay networks [22] and cloud analytics [4], using a variant of Datalog enhanced with *location specifiers* – a distinguished attribute of each tuple that denotes the node where the tuple is stored. These specifications can then be efficiently executed in a distributed query processor.

An oft-overlooked challenge is that of *securing* these distributed systems, particularly when their execution spans administrative boundaries. For example, achieving secure distributed execution may require authentication, encryption, and integrity-checking of inter-node communications, as well as intra-node access control.

Although it may be possible to achieve a high level of security for a particular deployment using a "one-size-fits-all" solution that imposes rigid cryptographic protocols and constructs, we argue that a *flexible* security framework is applicable to a much broader range of environments while providing the necessary level of security. For example, while sharing sensitive data between nodes on the Internet may require strong authentication and encryption mechanisms, *capabilities* may suffice within a trusted execution environment (e.g., a single datacenter [14]). The choice of security mechanisms may also depend on the user's estimation of the threat model in the current environment (e.g., assumptions about an attacker's abilities). Similarly, bandwidth and computation constraints (such as in sensor motes) may necessitate tradeoffs between resource consumption and security.

Reasoning about security in distributed systems is further complicated by the fact that security is a *cross-cutting concern*: decoupling it from a system implementation in a traditional imperative language is often challenging or impossible. But it is precisely this separation of concerns that is vital for customizable security. Additionally, security properties are often stated as safety properties – invariants that ensure something bad will never happen (e.g., "we will never accept a message with an invalid signature"). Adding new system functionality or security features may inadvertently negate existing safety properties.

To address the above challenges, this paper presents *SecureBlox*, a distributed query processor that allows security properties to be expressed and customized in a variant of the Datalog language. Our implementation of SecureBlox is a series of extensions to *LogicBlox* [20], an emerging commercial Datalog-based platform for enterprise software systems. SecureBlox extends Datalog with distributed execution, static meta-programming, and a familiar notion of database integrity constraints. The declarative, commutative nature of Datalog rules, along with SecureBlox's support for static meta-programming and integrity constraints, ensure that programmers do not have to worry about *where* to enforce a security property, but can instead focus on *what* high-level property to enforce and *how* to enforce it (for example, the *what* may be only accepting messages from other users we trust, and the *how* may be using RSA digital signatures). Additionally, SecureBlox's integrity constraints allow security properties to be monotonic under additions to the program's rules or constraints.

Specifically, this paper makes the following contributions:

SecureBlox: We present the design and implementation of SecureBlox, an extensible declarative system that unifies distributed query processing with security policy enforcement. SecureBlox extends *LogicBlox* [20], an emerging commercial Datalog-based platform for enterprise software systems. LogicBlox enhances Datalog with features such as integrity constraints, a static type system that guarantees conformance to certain constraints for all possible schema instantiations, and user-defined functions that can be integrated into query execution. SecureBlox leverages LogicBlox features in novel ways: integrity constraints assist in the expression of security properties, and user-defined functions implement custom encryption operators. SecureBlox further enhances LogicBlox with static meta-programming and support for distributed execution.

Static meta-programming: We enhance the query language of LogicBlox with *BloxGenerics* , a set of language features for static meta-programming–compile-time code generation through user-defined declarative rules. BloxGenerics enables security properties to be specified generally, for arbitrary predicates. BloxGenerics further allows certain correctness criteria on generated code to be specified as meta-constraints; the BloxGenerics compiler verifies that such criteria will always be met at compile-time. Independent of the contributions of SecureBlox, the BloxGenerics framework demonstrates the practical realization of meta-programmability in a commercial Datalog engine.

Customizable secure data applications: We demonstrate that SecureBlox expresses a wide variety of security properties for authentication, confidentiality, integrity, speaks-for, and restricted delegation. Using these primitives, SecureBlox supports a variety of distributed network applications, including (i) an authenticated declarative routing [23] protocol with encrypted advertisements and (ii) an authenticated/encrypted parallel hash join operation. In these applications, SecureBlox is used not only to execute the distributed queries, but also to specify and enforce the relevant security properties. We have developed an implementation of SecureBlox and evaluated the system on a local cluster based on the above applications. Our results demonstrate SecureBlox's ability to specify and implement a variety of secure distributed systems with different cryptographic schemes in a manner that enables the programmer to intelligently select the desired levels of performance and security.

2. BACKGROUND: LogicBlox

We first present an overview of the LogicBlox architecture. LogicBlox is a commercial Datalog-based platform for building enterprise-scale corporate planning and pricing applications. LogicBlox is currently used in several commercial decision automation applications, including retail supply-chain management [26], insurance risk management, and software program analysis [7, 8, 27].

The LogicBlox platform uses the *Datalog^{LB}* query language, and an evaluation engine (the runtime) that executes *Datalog^{LB}* programs over data stored in a LogicBlox *workspace*. Figure 1 illustrates the overall architecture of LogicBlox. We next provide a brief overview of the *Datalog^{LB}* language features, as well as the LogicBlox runtime and workspace.

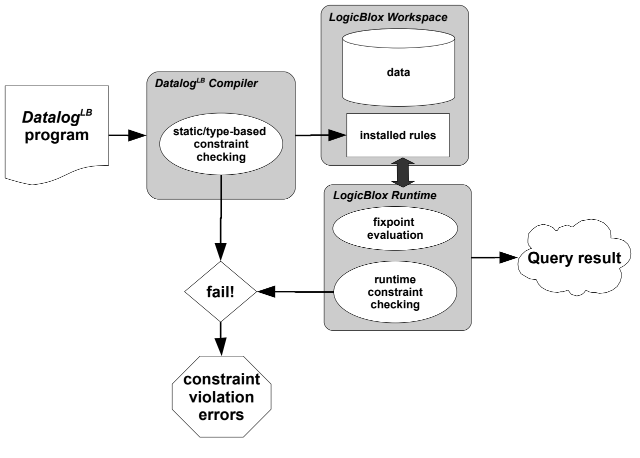

Figure 1: LogicBlox Architecture

The Datalog^{LB} Language. *Datalog^{LB}* extends ordinary Datalog with constructs for specifying integrity constraints such as functional dependencies, as well as a static type system that guarantees the validity of certain constraints for all schema instantiations.

The basic programming construct in *Datalog^{LB}* is "<-" (implication), which allows the declaration of a derivation rule:

```
q1, ..., qm <- p1, ..., pn.
```

The above rule means that "`q1` and ... and `qm` can be derived from `p1` and ... and `pn`". The `p`'s and `q`'s are *atoms*, which can either be predicates with variables or constants as arguments, comparison expressions (using relational operators such as <, =, etc), arithmetic expressions, or negated atoms. The left-hand-side of the <- is the *head* of the rule, and the right-hand-side comprises the rule's *body*.

Datalog^{LB} adds to Datalog the notion of an integrity constraint, declared using "->":

```
q1, ..., qm -> p1, ..., pn
```

Informally, the above constraint means that "if `q1` and ... and `qm`, then it must be true that `p1` and ... and `pn`". The difference between a constraint and a rule is that a rule derives data for the atoms in its head, whereas a constraint checks that for *existing* data on the left-hand-side of ->, the right-hand-side must hold. Whereas many constraints can only be effectively checked at runtime (e.g., foreign key violations), *Datalog^{LB}* employs a static type system, which guarantees at compile-time that certain kinds of constraints always hold for all possible instantiations of a given schema. The following is a typical type-based constraint, or a *type declaration*, for predicate `p`:

```
p(x1,...,xn) -> q1(x1), ... , qn(xn).
```

Unary predicates such as `qn` are considered "types". A value `x` is of type `q` if `q(x)` is true. The above constraint thus declares that for every tuple ⟨`x1`,...,`xn`⟩ in `p`, x_i must be of type q_i. The predicates `q1`,...,`qn` are referred to as the *types* of `p`. Type-based constraints can be checked at compile-time by verifying that every rule that derives facts for `p` implies the proper set membership for its arguments. For instance, the following rule will be rejected as not being type-safe, because the set of values in s is not guaranteed to be contained by the set qn:

```
p(x1,...,xn) <- ..., s(xn).
```

One way to make the above-rule type-safe is to declare that all elements of s is are guaranteed to be in qn, using a constraint: `s(x) -> qn(x).`

There are two language features that are used frequently in our case studies. First, *Datalog^{LB}* provides syntactic sugar for declaring a functional dependency for a predicate: `p[x1,...,xn]=y` declares that predicate p is a *function* from its first n arguments to the last argument. Functional dependencies are checked at runtime. Furthermore, a predicate can be declared as a *singleton*, i.e., a predicate with exactly one value (a constant). For instance, `p[]=v` declares p to be a singleton that contains only the value v. The value can be retrieved through `p[]`.

LogicBlox Evaluation Engine & Workspace. After compilation, a LogicBlox program – i.e., rules and constraints – is loaded into the *workspace*. A LogicBlox workspace is essentially a database instance that contains a set of predicate definitions[1], a set of *installed* rules (similar to *continuous queries*), as well as constraints. Within a designated workspace, the LogicBlox API allows an application to query and modify the data defined by the workspace. For example, the application may add or remove facts from predicates. When predicate data is modified, the installed rules are incrementally maintained using the *DRed* algorithm [15].

The LogicBlox engine evaluates rules using the semi-naïve algorithm until a fixed-point is reached—that is, until no more facts can be derived for the atoms in the heads of the rules. At the same time, the engine checks for constraint violations for every new fact that is derived, throwing constraint violation errors and restoring database consistency when appropriate.

3. OVERVIEW OF SecureBlox

Figure 2 shows an overview of the SecureBlox architecture and its usage model. The details of the LogicBlox platform are elided, as it is shown in detail in Figure 1. SecureBlox enhances LogicBlox with BloxGenerics and a distributed query processing runtime. These extensions are shown in light-grey boxes, labeled "SecureBlox".

SecureBlox takes as input user queries for data processing applications, as well as the security policies required by these applications. Queries, together with security policies, are input to the BloxGenerics compiler. The BloxGenerics compiler may throw compile-time errors for queries and policies that do not satisfy specified correctness criteria (details are presented in the next section). The BloxGenerics compiler then rewrites the queries and generates additional rules and constraints based on the security policies. The result of the BloxGenerics compilation is a set of modified queries, augmented with various security constructs.

These modified queries then go through the LogicBlox pipeline of compilation and loading into the workspace. The distributed query processing extension of SecureBlox then disseminates appropriate queries and security policies (out-of-band) to all participating nodes. During query execution, intermediate tuples are exchanged among nodes, and the query results are similarly assembled from various nodes at the end of the execution, or further distributed in the network based on the query specifications.

[1]A predicate definition declares both the logical attributes of a predicate, such as its name and arity; and also the physical attributes for the purposes of cost-based optimizations, such as the predicate's data storage format, data location, and population density statistics.

3.1 First Motivating Example

We motivate SecureBlox using the following example to highlight the need for customizable security policies, and outline how such customization can be achieved using SecureBlox:

```
reachable(X,Y) <- link(X,Y).
reachable(X,Y) <- link(X,Z), reachable(Z,Y).
```

The above rules define the transitive closure of the `link` predicate as `reachable`. In a distributed environment, each node may store different facts for `link`. For ease of exposition, we assume that the first argument of each fact in `link` (and thus `reachable`) is also the location of that fact. The above program is hence a distributed transitive closure computation. Prior work on declarative networking [21, 23] has shown that distributed transitive closure computations can be used as a basis for expressing routing protocols.

A central component of security in a distributed setting is *authentication*: where the identity of a principal is established and verified. Most previous work represents authentication using a distinguished keyword, `says`, that associates a principal with some fact. For instance, using the SeNDLog language introduced by Zhou *et al.* [29], the above query would be rewritten as follows, prefixing each body predicate with the remote principal asserting the fact:

```
reachable(X,Y) <- link(X,Y).
reachable(X,Y) <- link(X,Z), Z says reachable(Z,Y).
```

The `says` operator implements a common logical construct in authentication [18]: we assert "`p says s`" if the principal `p` supports the statement `s`. The `says` keyword enables the distribution system at principal `X` to authenticate the `reachable(Z,Y)` fact from `Z` before using it to derive its own `reachable(X,Y)` fact.

The notion of `says` is usually a hard-wired concept that is baked into the underlying system [1, 11]. However, a fixed policy over `says` can only work in a limited setting. For instance, users of the database have no way of applying different authentication schemes. Moreover, principals that receive facts do not have mechanisms for *authorization*, e.g., limiting how remote principals can modify local predicates. Nor do users have a choice over methods of encryption (if encryption is supported).

SecureBlox aims to support a heterogeneous execution environment in which nodes have different security levels. Thus, the users, not the underlying system, must be able to specify the appropriate security policies for their environment. In a hostile world, "`says`" may require digital signatures. In a more benign world, "`says`" may simply append a cleartext principal header to a message—and this will of course be cheaper. Somewhere in between, the use of digital signatures may be applied only to certain important messages. One may also want to use different cryptographic or non-cryptographic schemes. Next, we show how to declare and customize policies for `says` in SecureBlox. We provide additional security policy variants in Section 6, and use-cases of these policies in Section 7.

3.2 Customizable Policies Using SecureBlox

In SecureBlox, `says` can be implemented as a predicate-to-predicate mapping: given any predicate `T`, `says` maps `T` to a predicate `ST`, the "said" version of `T`. We can then define different policies and constraints on `ST` regarding its use. We define such a predicate mapping using BloxGenerics, the meta-programming facility we developed for LogicBlox. In the following examples, we informally explain the BloxGenerics rules by appealing to the reader's intuition. We detail the design and semantics of BloxGenerics in

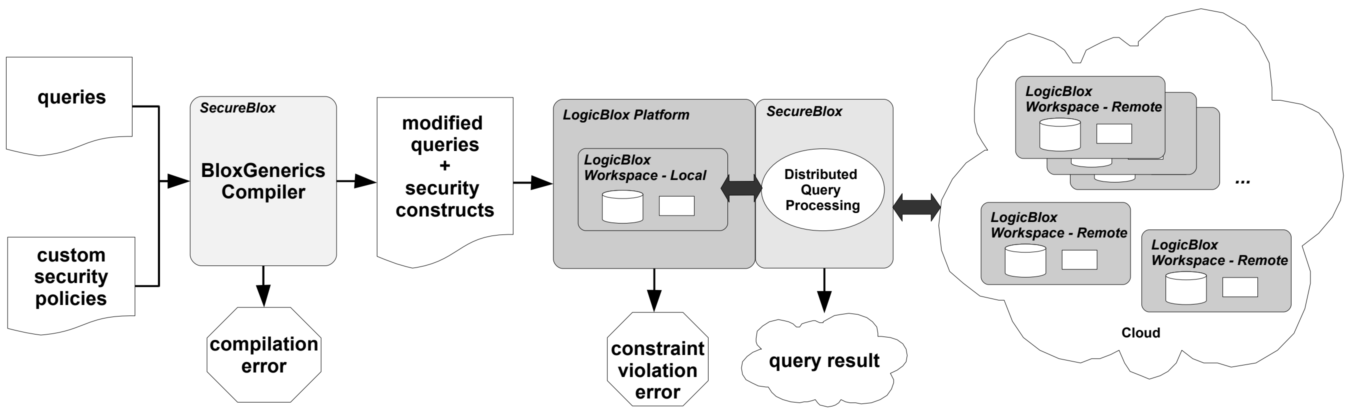
queries
custom security policies
SecureBlox
BloxGenerics Compiler
compilation error
modified queries + security constructs
LogicBlox Platform
LogicBlox Workspace - Local
constraint violation error
SecureBlox
Distributed Query Processing
query result
LogicBlox Workspace - Remote
LogicBlox Workspace - Remote
LogicBlox Workspace - Remote
...
Cloud

Figure 2: Overview of SecureBlox's components

Section 4. The following is a simple policy for `says`, involving only authentication:

```
says[T]=ST, predicate(ST),
‘{
   ST(P1,P2,V*) ->
      principal(P1), principal(P2), types[T](V*).
 }
   <-- predicate(T).
```

The BloxGenerics rule above (or just a *generic* rule, as indicated by the "<--" in place of "<-") declares that for all `predicate T`, a new mapping exists for `says`, and it maps `T` to `ST`, where `ST` is also a `predicate`. Furthermore, a *Datalog*^{LB} constraint, enclosed in `‘{...}`, is also derived/generated for each predicate `T`. The derived constraint requires that `ST`'s first and second arguments must be of type (or in the set of) `principal`; the remainder of its arguments–as indicated by the variable-length argument notation `V*`–must have the same types as the argument `types` of predicate `T`. Arguments such as `V*` allow the arity of `ST` to vary based on the arity of `T`.

We refer to predicates that store facts about other predicates as *generic predicates*. The generic predicate `says` can be *parameterized* by *any* concrete predicate to retrieve the "said" counterpart. For instance, `says[‘reachable]` parameterizes `says` with the predicate `reachable`, and the result is a "said" version of `reachable`. A corresponding constraint is generated, restricting the first two arguments of `says[‘reachable]` to `principals`.

Using the above definition of `says`, the distributed version of the transitive closure can be written as follows:

```
reachable(X,Y) <- link(X,Y).
reachable(X,Y) <- link(X,Z),
                  says[‘reachable](Z,self[],Z,Y).
```

Note that the second rule makes use of the "said" version of `reachable`. `says[‘reachable]` takes two `principals` as the first two parameters: `Z` is the principal that "said" the fact, and `self[]` is the singleton representing the local principal that is receiving the fact. Because the first two arguments of `says[‘reachable]` are declared to be of type `principal`, the LogicBlox runtime will verify `Z` and `self[]` against the constraint that they are known `principals`—a simple method of authentication.

In a benign world, where a principal trusts all other principals, he may derive a fact for predicate `T` for every `T` fact that was "said" to him:

```
‘{ T(V*) <- says[T](P,self[],V*). }
   <-- predicate(T).
```

In a more hostile environment, however, one might want to add authorization and encryption policies.

Authorization. For more control over `says`, one might want to declare that "if a principal `P1` wishes to say a fact about predicate `T`, then `P1` must have write-access to `T`". This policy can be easily added to any existing policy with the following BloxGenerics rule:

```
‘{ says[T](P1,P2,V*) -> writeAccess[T](P1). }
   <-- predicate(T).
```

The above rule says that, for all `predicate T`, generate the constraint that the first argument of `says[T]` (i.e., the principal saying the fact) must be part of the `writeAccess[T]` relation. `writeAccess` is simply another generic predicate that can be locally populated with principals allowed to write to a particular predicate.

Cryptography. In a more hostile environment, one may wish to cryptographically sign and verify facts exchanged between principals, as well as employ encryption to thwart eavesdropping. The following generic rule declares a constraint for every predicate `T`, requiring that every fact said about `T` by a principal `P` must have a valid signature (`sig`), as verified by `rsa_verify` given `P`'s `public_key`):

```
‘{
   says[T](P,self[],V*) ->
     sig[T](P,self[],V*,S),
     public_key(P,K), rsa_verify[T](K,V*,S).
 }
   <-- predicate(T).
```

Note that `rsa_verify` is implemented as a user-defined function. LogicBlox provides a set of APIs for hooking user-defined functions into rule or constraint execution.

Signature generation can be similarly defined with a generic rule, using a principal's `private_key`:[2]

```
‘{
   sig[T](self[],P,V*,S)
      <- says[T](self[],P,V*),
         private_key[]=K, rsa_sign[P](K,V*,S).
 }
   <-- predicate(T).
```

The above generic rule says that, for every predicate `T`, gener-

[2] In practice, we have found it useful to sign aggregates of serialized facts, rather than signing individual tuples, as a transaction may result in the transit of multiple tuples to a single node.

ate the rule that, for every fact about `T` by local node (`self[]`), produces the signature by `rsa_signing` the fact `V*` with the local `private_key`.

Alternate Cryptographic Scheme. A different encryption scheme can be easily constructed by redefining the rules for signature generation and verification. For instance, one can implement a keyed-Hash Message Authentication Code (HMAC), by replacing `public_key` and `private_key` with a shared `secret`, and replacing `rsa_sign` and `rsa_verify` with `hmac_sign` and `hmac_verify`:

```
‘{
   says[T](P,self[],V*) ->
     sig[T](P,self[],V*,S),
     secret(U,K), hmac_verify[T](K,V*,S).

   sig[T](self[],P,V*,S)
     <- says[T](self[],P,V*),
        secret(U,K), hmac_sign[T](K,V*,S).
 }
   <-- predicate(T).
```

We assume that `hmac_verify` and `hmac_sign` predicates have been implemented as user-defined functions.

The above examples give a flavor of how one can use SecureBlox to build customized security policies. Notice that in SecureBlox, the notion of `says` is defined as derivation rules and constraints – it is not a hard-wired concept. In previously proposed systems [1,11,29] where `says` is baked into the underlying runtime, it would take a very motivated programmer to change its meaning and interpretation. SecureBlox does not assume a specific security model–its goal is to be as general as possible so as to enable users to build constructs that reflect any security model of their choice. When a user programs or chooses constructs, she makes assumptions about an adversary, and makes trade-offs between security and efficiency.

Note that the ability to declare `says` as a generic predicate through meta-programming is crucial to the generality of SecureBlox. If one cannot declare `says` in a way that abstracts over the specific predicates it may be parameterized with, one would need to manually declare constraints involving `says` *for every predicate that can be "said"*. That is, one would need to declare constraints for not only `says_reachable`, but also `says_foo`, `says_bar`, etc. It is of course *impossible* to anticipate all possible predicates that may be defined and said. Thus, meta-programming is an indispensable language feature in supporting security policies, and in decoupling security concerns from the implementation of a system.

In the next two sections, we discuss in detail SecureBlox's extensions to LogicBlox: Section 4 discusses BloxGenerics, a meta-programming facility, and Section 5 discusses our distributed query processing mechanism.

4. BloxGenerics

Security policies in SecureBlox are essentially programs that specify rules and constraints over *Datalog*^{LB} programs. Hence, we refer to such programs as *meta-programs*. SecureBlox achieves static meta-programmability using *BloxGenerics*, a language mechanism that allows rules and constraints to be declared over structural elements of *Datalog*^{LB} programs: predicates, rules, constraints, formulas, etc. Furthermore, *Datalog*^{LB} programs can be generated using BloxGenerics rules. While meta-programming mechanisms exist for other Datalog dialects [10] as well as Prolog [9], BloxGenerics distinguishes itself by guaranteeing that generated code will always obey the correctness criteria of the *Datalog*^{LB} language, as well as those specified by the programmers.

4.1 BloxGenerics Language Features

BloxGenerics adds four language constructs to *Datalog*^{LB}: `<--` for declaring generic rules, `‘{...}` for declaring code templates, a notation for declaring variable-length variable sequences, and `-->` for declaring generic constraints. In the rest of this section, we discuss these language features using the following BloxGenerics rule presented in Section 3:

```
says[T]=ST, predicate(ST),
‘{
   ST(P1,P2,V*) ->
     principal(P1), principal(P2), types[T](V*).
 }
   <-- predicate(T).
```

4.1.1 Generic Rules

A generic rule specifies how facts about *Datalog*^{LB} program elements (e.g., predicates, rules, etc.) can be derived from other facts about *Datalog*^{LB} program elements. For instance, the above generic rule defines how new predicates (`ST`) can be derived from existing predicates (`T`).

`<--` has very similar semantics to the regular rule implication arrow, `<-`, for *Datalog*^{LB}. The right-hand-side of `<--` (the body) specifies the conditions for the rule's firing; the left-hand-side (the head) specifies the relations derived as a result. The difference, however, is that whereas `<-` rules define relationships between values such as strings, integers, etc, the only predicates allowed in generic rules are generic predicates—i.e., predicates that store relationships between program elements.

BloxGenerics provides a number of built-in generic predicates to facilitate declaration of BloxGenerics rules: `predicate(p)` is the set of predicates; `rule(r)` is the set of rules; `ruleHead(r,h)` is a relation between a rule and the formula comprising its head; and `atom(a)` is the set of atoms. Additionally, programmers can declare custom generic predicates. For instance, `says` is a user-defined generic predicate. A BloxGenerics compiler pipeline stage converts input *Datalog*^{LB} programs into their relational representations and populates these generic predicates.

Head-Existentials. Note that a generic rule allows variables that are not bound in its body to appear in its head. For instance, `ST` is not bound by any predicate in the body of the above rule. This type of variable is referred to as being *"existential in the head"* of the rule. For each head-existential variable, a new entity must be created for each fact in the body. For our use cases, this is precisely the desired behavior. For instance, for every predicate `T`, we want a new predicate `ST` to be created to represent the "said" version of `T`.

The admittance of head-existential variables, however, takes BloxGenerics beyond Datalog's P-time termination guarantees. Indeed, the evaluation of a BloxGenerics rule may *not* terminate with a fixpoint. There have recently been a number of breakthroughs on termination conditions for Datalog rules with head-existentials [12, 25], mostly in the context of the data exchange problem. Exploring these techniques, as well as developing our own solutions for strong termination guarantees, is part of our future work. The current BloxGenerics compiler throws a compiler error if no fixpoint is reached within a time limit. However, we note that none of the BloxGenerics rules or constraints presented in this work result in a non-terminating compilation phase, and we stress that the issue of non-termination is restricted to the compilation phase: a local procedure that occurs strictly before execution of the system.

4.1.2 Code Templating

`‘{...}` is a templating mechanism for specifying the code to be derived/generated from generic rules. It allows a piece of *Datalog*^{LB}

code to be specified using *variables* in place of concrete predicate names. This quoting mechanism affords users the ability to specify security rules that apply to *any* predicate rather than to a specific concrete predicate. This level of generality is a necessity for maintaining a separation between security policy and system implementation.

`‘{...}`, however, is simply syntactic sugar for fully specifying the structure of the templated code using generic predicates. For instance, `‘{ ST(P1,P2,V*) -> principal(P1), ...}`, is syntactic sugar for the following formula:

```
constraint(C1), lhs(C1,H1), atom(H1), atomPred(H1,ST),
body(C1,B1), atom(B1), atomPred(B1,principal), ...
```

The BloxGenerics compiler expands every quoted rule into a full specification.

4.1.3 Variable-length Argument Sequence

The above example makes use of a feature for declaring a predicate with a variable number of arguments. Note that `ST` has argument `P1`, `P2`, and `V*`. The `*` notation indicates that argument `V` is a sequence of variables of any length, including length 0. The length of `V*` is bound by the `types` of `T`. Depending on which concrete predicate instantiates `T`, `V*` may instantiate to arbitrary length variables. For instance, since `reachable` is a predicate of arity 2, `V*` expands to “`V0,V1`” for `says[‘reachable]`.

4.1.4 Generic Constraints

One of the distinguishing features of BloxGenerics is that it allows programmers to specify the necessary correctness criteria for generated code using *generic constraints*. The compiler guarantees that all possible code generated from a template will obey the specified constraint before the actual (run-time) code generation.

For instance, one may want to restrict which predicates’ data are allowed to be “said”. This constraint can be easily defined as a generic constraint on `says`:

```
says(P,SP) --> exportable(P).
```

Note the use of `-->`, the constraint arrow for BloxGenerics. The above generic constraint says that for all tuples `says(P,SP)`, `P` must be in the `exportable` relation (`exportable` can be separately declared and have the appropriate predicates inserted into it).

The BloxGenerics compiler will check every rule producing `says` tuples to verify whether such a constraint will be followed. In fact, for the example rule in this section, this constraint does not always hold, and the BloxGenerics compiler will reject the program for this reason. The fix is to simply add a condition in the body of the generic rule, restricting `T` to those predicates in `exportable`:

```
says(T,ST), predicate(ST), ‘{ ... }
   <-- predicate(T), exportable(T).
```

4.2 Compilation of BloxGenerics Programs

Figure 3 shows a detailed view of the BloxGenerics compiler architecture. The compilation of a BloxGenerics program is essentially the evaluation of BloxGenerics rules and the verification of BloxGenerics constraints. As we have shown, a BloxGenerics rule or constraint is simply a *Datalog*^{LB} rule or constraint (with support for head-existential variables) that computes over a *Datalog*^{LB} program as data. Thus, the BloxGenerics compiler evaluates generic rules (and checks constraints) using the same engine that evaluates regular *Datalog*^{LB} rules.

First, a relational representation is created for the incoming query. That is, the structure of the query is represented using predicates such as `predicate`, `rule`, etc. Next, the generic rules and the queries

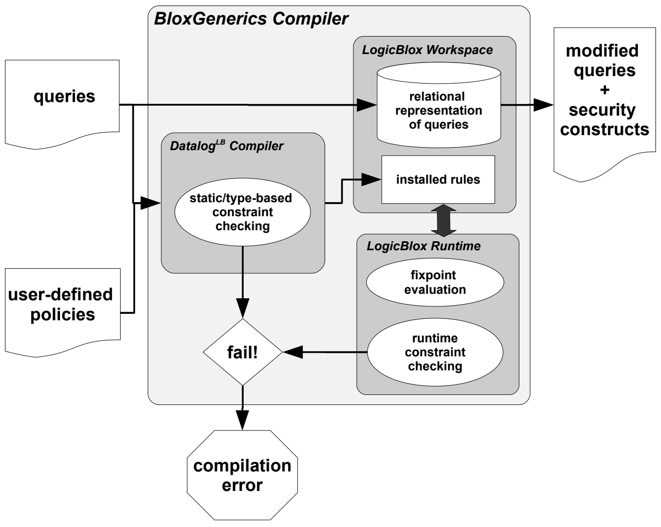

Figure 3: Detailed View of BloxGenerics Compiler

go through a static type-based constraint checking process. The generic rules are then evaluated over the program data. The evaluation of generic rules may cause more facts to be derived, and thus, more *Datalog*^{LB} programs to be generated (e.g., rules about a specialized version of `says` predicate may be defined). When a fixpoint is reached, the BloxGenerics compiler reifies a *Datalog*^{LB} program from the workspace. This modified version of the input query is the output of the BloxGenerics compiler.

Note that the BloxGenerics compiler in Figure 3 is a subset of the LogicBlox compiler/runtime in Figure 1. Our implementation bootstraps the BloxGenerics compiler using the LogicBlox compiler, converting *Datalog*^{LB} programs to data, handling head-existentials, and reifying *Datalog*^{LB} programs from their relational representations.

5. DISTRIBUTED EXECUTION

In this section, we describe language support for expressing distribution in *Datalog*^{LB} and discuss how distributed execution is handled in the presence of constraint checks.

5.1 Principals and Nodes

SecureBlox supports distributed computation by enabling users to partition predicates by *location*. When a node derives a fact at a location other than itself, the fact is implicitly communicated to the correct node via a UDP message. SecureBlox represents location with the built-in `node` type. An element of the `node` type consists of an IP address and port that specifies the node’s location. A predicate is partitioned by location if its first argument is `node`. Any existing predicate, `p(X1, ..., Xn) -> t1(X1), ..., tn(Xn).` may be partitioned by creating a predicate `p'` with a `node` argument in its first key:

```
p'(N, X1, ..., Xn) -> node(N), t1(X1), ..., tn(Xn).
```

Furthermore, a rule encapsulating the *placement policy* of `p'` needs to be specified. For example, one may wish to define a mapping from the first key attribute of `p` to the set of nodes, `x1node`, and use it to populate the partitions of `p'`:

```
p'(N, X1, ..., Xn) <- p(X1, ..., Xn), x1node[X1]=N.
```

In addition to nodes, SecureBlox provides a notion of *principals* that are independent of physical nodes. A principal may be identified by a global value. Specifying policies in terms of principals rather than nodes allows policies to be *location transparent*. A principal is temporarily associated with a node through the `principal_node` mapping, which stores a single node per principal, representing the principal's location. The local principal is represented by the built-in `self[]` singleton predicate.

For example, the `says` construct introduced in Section 3.2 is defined in terms of principals. Exportation of `says` facts and signatures is handled by the following export rule which `serializes` a said fact along with its signature, looks up the corresponding node `N` for the principal specified in `says`, and exports the fact to `N` by deriving `export` at `N`'s location:

```
‘{
   export(N, L, T)
     <- says[P](self[], U, V*),
        sig[P](self[], U, V*, S),
        serialize[P](S, T, V*),
        principal_node[U] = N,
        principal_node[self[]] = L.
 }
   <-- predicate(P).
```

At the destination, `says` facts and signatures are deserialized and imported using a similar rule:

```
‘{
   says[P](U, self[], V*)
     <- export(N, L, T),
        deserialize[P](S, T, V*),
        sig[P](U, self[], V*, S),
        principal_node[U] = N,
        principal_node[self[]] = L.
 }
   <-- predicate(P).
```

To prevent unauthorized principals from interpreting facts, one can expand the purview of the `says` construct to encompass encryption. Implementing this requires the following simple modifications to the export rule (and a similar modification to the import rule). To utilize AES, one applies the `aesencrypt` operator on the serialized text (making use of a shared `secret`, `K`, with node `N`) and `exports` the encrypted text:

```
‘{
   export(N, L, CT)
     <- says[P](self[], U, V*),
        sig[P](self[], U, V*, S),
        serialize[P](S, PT, V*),
        principal_node[U]=N,
        principal_node[self[]]=L,
        secret(N, K), aesencrypt(PT, K, CT).
 }
   <-- predicate(P).
```

Note that the only difference between the first `export` rule in this Section to the rule above is the last line in `‘{...}`, where we reference the `aesencrypt` operator and the `secret` predicate.

5.2 Distributed Evaluation with Constraints

Constraints evaluated at runtime set conditions under which program evaluation should fail (for example, if a signature associated with a `says` tuple is invalid). SecureBlox enforces constraints by processing a batch of incoming facts from a remote node in a local ACID transaction that encapsulates a fixpoint computation. If a derivation in the transaction violates a runtime constraint, then the transaction (including the input tuples) are rolled back. Derivations are committed to the workspace and network messages are sent only after a transaction succeeds.

Our implementation contrasts with the *pipelined semi-naïve* (PSN) evaluation method used for executing distributed Datalog in declarative networking [21]. In PSN, tuples are processed tuple-at-time in a pipelined fashion. Each node maintains a FIFO queue (ordered by arrival timestamp) of new input tuples that can either be incoming network or local events. Each new tuple is dequeued and used as input to local rules. The execution of a rule may generate new tuples that are either inserted into the local queue or sent to a remote node for further execution. Duplicate evaluations are avoided using local arrival timestamps; each new tuple is combined only with older tuples.

The advantage of PSN over local transactions in our implementation of SecureBlox is that PSN may send tuples to remote nodes before a fixpoint is reached, thus enabling lower latency to the first output from a distributed computation, and faster convergence. However, PSN's default behavior in this regard makes it unsuitable for supporting local constraints. For example, PSN may send a tuple whose local transitive consequences later result in a local constraint violation. Without distributed transactions, it is not obvious how the node should retract this tuple when it learns of the constraint violation.

One possible enhancement to SecureBlox that we are exploring for future work is to relax local transactions: instead of sending all results at the end of a local transaction, we could send results in a predicate `P` as soon as all integrity constraints involving `P` – and any predicate upon which `P` is transitively dependent – have been evaluated. This approach would allow us to achieve lower latency for the first output while guaranteeing the integrity of all constraints.

6. SECURITY MECHANISMS

We have shown in Section 3 how different security policies involving distributed data processing can be implement using rules and constraints over the predicate `says`. In this section, we build upon those concepts, and show two more example policies: *trust delegation* and *anonymity*.

6.1 Trust Delegation

When a principal does not wish to trust all other principals, he may selectively delegate trust by only accepting facts from certain other principals. For example, the following generic rule expresses that, if a principal `P` from a set of `trustworthy` principals says a fact about any predicate `T` to me, then I will accept that fact into my local predicate `T`:

```
‘{
    T(V*) <- says[T](P,self[],V*), trustworthy(P).
 }
    <-- predicate(T).
```

A principal may also perform delegation on a more fine-grained level. Consider, for example, a per-predicate delegation policy. If a principal trusts a different set of nodes for each predicate, we can declare `trustworthyPerPred` as a generic predicate such that each `trustworthyPerPred[T]` contains the set of principals trusted for predicate `T`:

```
‘{
    T(V*) <- says[T](P,self[],V*), trustworthyPerPred[T](P).
 }
    <-- predicate(T).
```

If a principal wishes to trust a credit agency `CA` about `creditscore`, this information can be declared locally as:

```
trustworthyPerPred[‘creditscore]("CA").
```

If he wishes to ensure that creditscore is delegated to CA (and no other principals), he may declare the constraint:

```
trustworthyPerPred[‘creditscore](U) -> U = "CA".
```

Note that there are other notions of delegation, such as allowing another principal to act with your authority. SecureBlox can easily express this type of delegation, but we omit those rules for brevity.

6.2 Anonymity

A low-latency anonymity system, like Tor [13], enables anonymous unicast between two hosts on the Internet through obfuscation of a sender's (and optionally a receiver's) Internet Protocol (IP) address by relaying communication through an *anonymous path* – a series of intermediate nodes. Users of anonymity systems include dissidents, who can freely blog their opinions without fear of retribution, law enforcement, who can conduct undercover operations, and privacy-conscious users, who can avoid tracking. The Tor website maintains a comprehensive set of anonymity use cases [28].

Communicating using Tor first involves an initiator securely establishing forwarding state and keys at each intermediate node on the path, so that each node knows only its predecessor and successor in the path. Thus, no single node can discern the identity of the initiator with high probability. Interestingly, these path instantiation rules can be expressed in SecureBlox, but we omit them for brevity. We instead focus on how one would build an anon_says construct to support anonymous communication. To send a message over an instantiated anonymous path, an initiator repeatedly encrypts the message with keys shared with each node in the path, and forwards this multiply-encrypted message to the first node in the anonymous path. As each intermediate node sees the message, it removes the layer encrypted with its key and forwards the message onward. The rule below shows the initial step:

```
‘{
   anon_export(N, Id, CT)
     <- anon_says[P](self[], U, V*),
        anon_serialize[P](T, V*),
        anon_path[U] = C,
        anon_path_forward_id[C] = Id,
        anon_path_nexthop[C] = N,
        anon_encrypt(C, T, CT).
 }
   <-- predicate(P).
```

The above rule serializes[3] and iteratively encrypts (using anon_encrypt) any outgoing anon_says fact, and sends it via anon_export to the first hop N of the appropriate anon_path C. Note that anon_export contains Id, a link-local identifier. This value is used to look up the appropriate next hop in the anonymous path at an intermediate node.

An intermediate node receives an anon_export message, looks up the link local identifier sent by the previous node, and exports the message to the corresponding next hop N2 (with link identifier Id2) after peeling off one layer of the encryption using anon_decrypt:

```
anon_export(N2, Id2, CT2)
   <- anon_export(self[], N1, Id, CT1),
      anon_path_backward_id[C] = Id1,
      anon_path_forward_id[C] = Id2,
      anon_path_nexthop[C] = N2,
      anon_decrypt(C, CT1, CT2).
```

At the endpoint of the anonymous path, the following rule is triggered, which derives an incoming anon_says_id_in fact. Note the asymmetry between sending and receiving: the initiator builds a circuit and thus knows the identity of the endpoint; however, the endpoint does not know the identity of the initiator, and can only refer to him via the circuit C.

```
‘{
    anon_says_id_in[P](C, V*)
       <- anon_export(self[], N1, Id, CT1),
          anon_path_backward_id[C] = Id,
          anon_path_endpoint[C] = true,
          anon_decrypt(C, CT1, PT),
          anon_deserialize[P](PT, V*).
 }
   <-- predicate(P).
```

Additional logic is required for forwarding backwards along the circuit (from the endpoint to the initiator), however we omit this due to space constraints.

7. USE CASES

We next outline three use cases: a path-vector routing protocol, a parallel hash join, and a distributed join over an anonymizing network. Note that our use cases are independent of the implementation of our security constructs (says, anon_says, trustworthy, etc); our use cases may employ any implementations of security constructs, not just those presented in previous sections.

7.1 Path-Vector Protocol

A *path-vector* routing protocol is a distributed all-pairs-shortest-path computation. The computation joins links (paths of length one) with paths of length at least one to form new paths which are propagated to neighbors. The protocol propagates the entire composition of each path so nodes can make policy decisions about whether to accept the path and when to route traffic through it. For example, if nodes sign the path as it is built, then a router may have the constraints "require that accepted paths are from some set of routers that I trust" or "require that no accepted paths route through node X."

Below, we present a schema (or type declarations) for the path vector computation:

```
path[P,Src,Dst]=C -> pathvar(P), node(Src), node(Dst),
                     int[32](C).
pathlink[P,H1]=H2 -> pathvar(P), node(H1), node(H2).
bestcost[Src,Dst]=C -> node(N1), node(N2), int[32](C).
pathvar(P) -> .
link(N1,N2) -> node(N1), node(N2).
```

Intuitively, a path from a source (Src) to a destination (Dst) has an associated cost (C) and consists of a series of links (pathlink) between two hops (H1 and H2). As there may be many paths between two nodes, the bestcost predicate stores the cost of the best path. The cost C may represent any metric on which a protocol makes forwarding decisions–in our case, it represents the number of hops between the source and the destination.

We use the pathvar entity to relate a path with its series of pathlink tuples. This allows us to avoid introducing an explicit set or list type to Datalog (and the associated termination issues), as in [23].

The first rule is the base case of the distributed transitive closure computation–it expresses that whenever there exists a link from Me to N, then let there exist a path of length one from Me to N:[4]

[3] anon_serialize represents serialize without the signature argument – it would be detrimental to a principal's anonymity for her to identify herself as the author of the message.

[4] We use the head-existential variable P for ease of exposition. Because a routing protocol specifies one best path between a given source and destination, we could instead modify the protocol and replace P with two variables of type node representing the source and destination of the path.

```
pathvar(P),
path[P, self[], U]=1,
pathlink[P, Me]=N
  <- link(Me,N),
     principal_node[self[]]=Me,
     principal_node[U]=N.
```

The following rule joins a link fact with a path of best cost, and says the resulting path and pathlink tuples to the linked node. The rule reduces message exchange by not advertising a path to a node if that node already appears in the path:[5]

```
says['path](self[], U, P, N, N2, C + 1),
says['pathlink](self[], U, P, H1, H2),
says['pathlink](self[], U, P, N1, Me)
  <- pathlink[P, H1]=H2, link(Me, N), path[P, Me, N2]=C,
     bestcost[Me, N2]=C,
     principal_node[U]=N,
     principal_node[self[]]=Me,
     N != N2, !pathlink[P,N]= _.
```

The bestcost tuple may be computed through a min aggregate as shown below, using the LogicBlox syntax for aggregation:

```
bestcost[Me, N]=C
  <- agg<< C=min(Cx) >> path[Me, N, _]=Cx.
```

7.2 Parallel Hash Join

In a system where tuples are partitioned by a hash function on some key attribute (e.g. a DHT [16]), performing a join on an attribute other than the key involves *rehashing* the tuples on the join attribute, joining the tuples, and sending the results to the initiator.

For example, if table A is hashed on its first key attribute, the following rules rehash it on its second key attribute:

```
says['A](self[],U,e1,e2)
  <- A(e1,e2), sha1(e2,hash),
     prin_maxhash[U] = max, max > hash,
     prin_minhash[U] = bot, min >= bot.
```

In the above rule, prin_minhash and prin_maxhash represent the range of hashes that a given principal is responsible for storing. The rule performs a hash on the second key of A tuples, looks up the principal U whose range contains this value, and says the tuple to U. A similar rule exists for table B.

The following rule performs a join on tables A and B based on their second attributes, sending the result back to the initiator:

```
says['AB_joinresult](self[],u,e1,e2,e3)
  <- A(e1,e2), B(e3,e2),
     initiator[]=u.
```

Building upon the above rules, one can perform two types of authentication by simply modifying the says definition as presented in the previous sections: In *node-level authentication*, machines can be authenticated. For instance, in a cloud consisting of untrusted machines, each machine can authenticate all incoming rehashed tuples to ensure that only tuples sent from trusted machines are used in the parallel hash join operation. Communication among machines can be additionally encrypted.

In *user-level authentication*, the initiator needs to be authenticated and checked against the access control policies to ensure that she has the right permissions across tables A and B. This can be enforced via a constraint that checks for read/write permissions on table A and B, in a similar fashion as presented in Section 3.2.

7.3 Anonymous Join

We consider a special case where an anonymous user wants to join a small local table with a publicly available large, remotely located table—for example, joining a local table of topics she is interested in (interests) with a large public repository of information (publicdata). We want to avoid transferring the entire table to the initiator, or the initiator transferring his entire table to the remote node, as bandwidth is a precious commodity in anonymity networks. Running a hash join between the two nodes may be a better solution. Composing a hash join with our earlier anonymous_says primitive for anonymous communication is easy in SecureBlox. The join involves an initiator performing a hash on the join key of his interests table, and anonymously saying a req_publicdata tuple to the table_owner of publicdata:

```
anonymous_says['req_publicdata](self[], U, Hx, 1)
  <- interests(X,_),
     table_owner['publicdata](U),
     sha1(X,Hx).
```

The owner of publicdata executes the rule below to relay all tuples with the matching hash back along the anonymous path they arrived on:

```
anonymous_says_id_out['publicdata](C, X, Y)
  <- publicdata(X,Y),
     anonymous_says_id_in['req_publicdata](C, Hx, 1),
     sha1(x,hx).
```

8. EVALUATION

We evaluate the SecureBlox prototype on a local cluster using two of our use cases – the path vector protocol, and the parallel hash join. All our experiments are carried out on a 36 node cluster, interconnected by Gigabit Ethernet. Nodes in the cluster have CPUs ranging from dual-core 2.80GHz Pentium Ds to 2.83GHz quad-core Xeons, and run Fedora 10 with Linux Kernel 2.6.27.

8.1 Path Vector Protocol

For the path vector protocol, we evaluate the following metrics:

Fixpoint latency (s): The time taken for a SecureBlox program to reach a distributed fixpoint based on the input network. A fixpoint is reached when no new facts are derived by any node in the system.

Per-node communication overhead (KB): The per-node bandwidth utilization for executing a SecureBlox program to a distributed fixpoint in a stable network, measured by averaging across the aggregate bandwidth of each node for the experiment's duration.

Average transaction duration (ms): The average length of a local SecureBlox ACID transaction averaged over all nodes for all experiments of a given graph size. The transaction duration represents the amount of time required to process a batch of incoming tuples, compute a local fixpoint, and export any tuples to other nodes.

Cumulative fraction of converged nodes: The fraction of nodes that do not process or receive any additional tuples after a certain time. When a node stops receiving and processing additional tuples, it has calculated the shortest path to all other nodes in the network.

We execute the protocol on network sizes ranging from 6 to 72 instances, where each physical cluster node executes one or two SecureBlox instances. As input to the path vector protocol, for each network size, we generate ten random graphs with an average node degree of three. Each data point represents the average over ten runs. We distribute initial links to all nodes simultaneously.

SecureBlox instances exchange messages over UDP using various combinations of authentication and confidentiality. We emphasize that customization of the encryption and digital signature

[5] An _ is Datalog convention for specifying a variable that appears only once in the body of a rule.

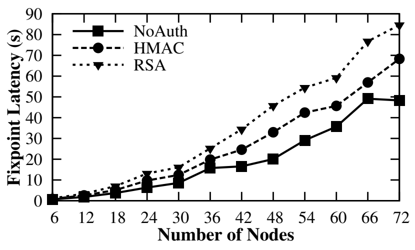

Figure 4: Fixpoint Latency (s) with no encryption.

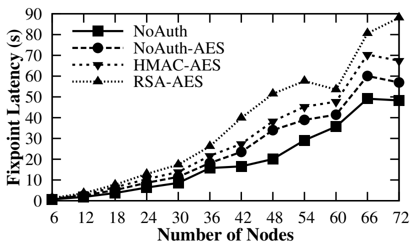

Figure 5: Fixpoint Latency (s) with encryption.

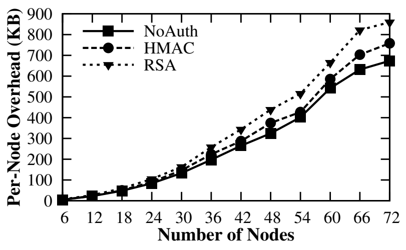

Figure 6: Aggregate communication overhead (KB) with no encryption.

schemes does not entail modifying the path-vector protocol specification – only minor changes to the implementation of the says construct are required.

Figures 4 through 6 compare fixpoint latency and communication, respectively, imposed by different methods of authentication and encryption. The signature schemes we evaluate are: no authentication (*NoAuth*), keyed-hash message authentication codes based on pairwise shared secrets (*HMAC*), and RSA digital signatures (*RSA*). Additionally, we apply either no encryption, or (*AES*) symmetric key encryption. HMAC derives a signature by applying a hash function – SHA-1 in our case – to a combination of the pairwise shared secret with the message. RSA authentication signs a SHA-1 digest of the data with the private key of the sender. We use 128-bit random shared secrets for HMAC and AES, and a 1024-bit keysize for RSA.

We make the following observations. First, in terms of communication overhead (Figure 6), *NoAuth* incurs the least overhead, as expected. For instance, when the network size is 36, *NoAuth* incurs an average per-node communication overhead of about 197 KB, as compared to 223 KB for *HMAC* and 258 KB for *RSA*. As expected, *HMAC* causes additional traffic compared to *NoAuth* due to the 20 byte output of the SHA-1 hash function. *RSA* incurs the highest overhead due to its 256 byte signatures per message. Second, a similar trend is observed when measuring fixpoint latency (Figures 4 and 5). In a 36 node network, we observe that when no authentication is used, the convergence time is approximately 15 seconds, whereas HMAC requires 19 seconds, and RSA needs 25 seconds to reach a fixpoint. Adding AES encryption to RSA signatures increases the fixpoint latency to 26 seconds. Our results are comparable to benchmarks presented in the SeNDLog work [29]. We believe that fixpoint latency will decrease as we relax the ACID transactions currently executed per-message at every node (Section 5.2).

Figure 8 shows the cumulative fraction of the nodes that have *converged* at any number of seconds after the start of the experiment, from a representative run of the path vector protocol on one of the 36 node random graphs. As expected, increasing the computational overhead of authentication causes a delay in the convergence of the first node (a right-shift of the graph) and a decrease in slope due to increased transaction duration. All lines on the graph exhibit a step-like behavior, caused by bursts of nodes converging. It is likely that these represent the various iterations of the shortest path computation. For example, all paths of length one are computed first, followed by paths of length two, etc. Certain groups of nodes may share certain longest shortest-path lengths, and thus converge simultaneously. Due to the larger number of possible shortest-path lengths in the 72 node case, we observe more, but smaller steps (Figure 9).

Figure 7 shows the average execution time of a transaction for *NoAuth*, *HMAC* and *RSA-AES*. We speculate that the initial decline in average transaction time is because nodes 1-8 use older Pentium D CPUs. When running two instances on nodes 1-8 (for configurations 36 nodes and larger) we see a jump in the transaction time for the computation-heavy RSA-AES scheme. Additionally, as the number of nodes increases, one expects to see slightly longer transaction times, as each transaction involves joining links with more paths. The large growth after 48 nodes was caused by several outlier experiments with high fixpoint execution time, possibly the result of a relatively uniform distribution of longest shortest-path lengths among the nodes – the cumulative fraction of converged nodes for these outlier experiments contained relatively few bursts of nodes converging (e.g. the step behavior observed in Figure 8 was far less pronounced). We omit the graph for brevity.

8.2 Secure Hash Joins

For the secure hash join, we evaluate per-node communication overhead on various experiment sizes. Additionally, we measure:

Cumulative fraction of transaction completion time: The fraction of transactions at the initiator of the join that are completed by a given time.

Our secure hash join algorithm performs a binary equi-join of two predicates of roughly equal size – one containing 900 tuples, and the other containing 800. Tuples are initially hashed on their first key attribute, so nodes must rehash on the second key attribute, join the tuples, and send the result to the initiator of the join. Our measurements commence at the beginning of the rehashing, and conclude when the initiator receives the last batch of joined result tuples. The second key attribute of each table is drawn from a domain consisting of 72 distinct join values (randomized for every trial). For each distinct join value, there are a roughly equal number of tuples, and it is expected that the distribution of rehashed tuples is roughly equal across all nodes. We ran ten trials per experiment size.

Figure 10 illustrates the cumulative fraction over all six-node experiments of transaction completion time at the initiator. Since our implementation of SecureBlox does not send any tuples over the network until the end of a local transaction (Section 5.2), there is a considerable lag until the first results arrive at the initiator. Because the batches are large with low parallelism, cryptographic overhead is low, as senders can perform a single signature operation per batch of tuples destined for a given node, and receivers can perform a single verification per batch. Figure 11 shows the same result over all 18-node experiments. The cryptographic overhead is more profound as the parallelism increases, because each sent and received batch of tuples is smaller. Thus, senders and receivers must perform more cryptographic operations.

Figure 12 shows the per-node overhead in kilobytes in relation to the size of the experiment. As expected, greater parallelism implies less per-node overhead. However, as the experiment size gets larger, we see diminishing returns due to (i) increasingly small mes-

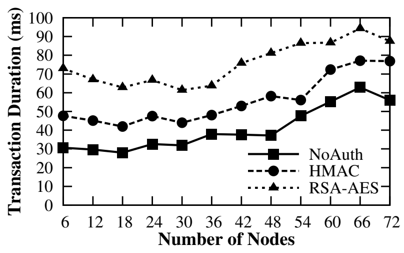

Figure 7: Average transaction duration (ms)

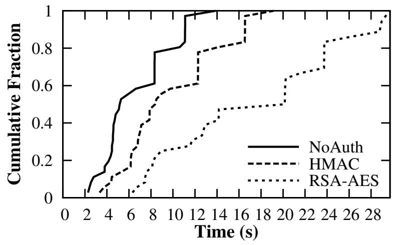

Figure 8: Cumulative fraction of converged nodes for one 36-node random graph.

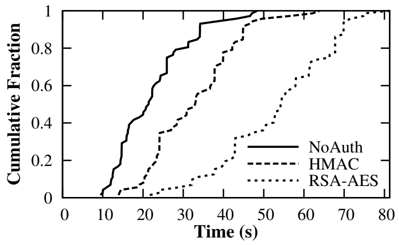

Figure 9: Cumulative fraction of converged nodes for one 72-node random graph.

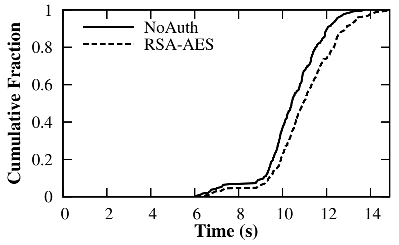

Figure 10: Cumulative fraction over all 6-node experiments of transaction completion time at the initiator.

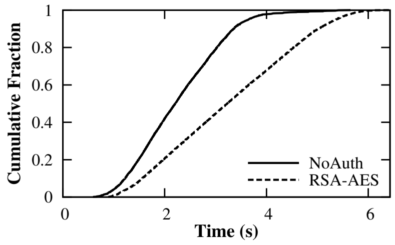

Figure 11: Cumulative fraction over all 18-node experiments of transaction completion time at the initiator.

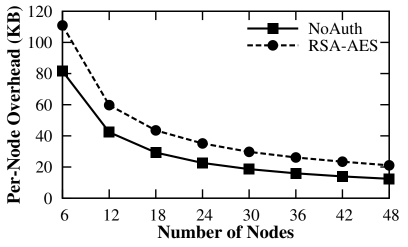

Figure 12: Per node communication overhead for secure hash join.

sages being exchanged, and (ii) the presence of two SecureBlox instances on some nodes.

9. RELATED WORK

SecureBlox's distributed query processing feature is inspired by *declarative networking* [21]; *Datalog^{LB}* is closely related to their *Network Datalog* (NDlog) language. Declarative networking views network protocols as distributed recursive queries over database relations (partitioned by location). Network communication is implicit when a fact is derived in another partition, but a trusted execution environment is assumed, and no attempt is made to enable security. NDlog has no notion of integrity constraints or meta-programming – these differences affect query execution, as explained in Section 5.

SecureBlox is inspired by recent work in *trust management* systems, which enable the formulation of access control policies and security credentials, determine whether particular credentials satisfy relevant policies, and support deferring trust to third parties (delegation) [6]. Recently, the security community has proposed several declarative trust management frameworks [1, 11, 17, 19]. These systems expose a small fixed set of hard-coded security operators, such as `says`. Thus, they do not enable reconfigurability or allow users to express custom security properties. For example, one may want to ensure that a user `Alice` can never access a resource `R`. This might entail locating all rules in the program that grant access to `R`, and inserting a restriction in each rule body that prevents `Alice` from accessing `R`. In contrast, SecureBlox's support for meta-programming and integrity constraints allows one to concisely express such a security property anywhere in the program.

Zhou *et al.* [29] propose the SeNDLog language, which adds the `says` operator to NDlog. Similarly to existing trust management systems, adding security constructs, and specifying custom policies, is difficult. Reconfiguration of their `says` operator necessitates modification and recompilation of their runtime engine.

Our earlier position paper [24] further proposes that extending Datalog with *dynamic meta-programming*, *meta constraints* (restrictions on the set of allowed programs), and a cryptographic library may enable one to customize the type of cryptography used for `says`, as well as the manner in which *delegation* restrictions are enforced. However, the paper does not specify an algorithm for evaluating the dynamic meta syntax that it introduces. Also, the proposed addition of dynamic (runtime) meta-programming to Datalog potentially makes the language unsafe by enabling one to write non-terminating programs and by allowing derivation of non-typesafe rules at runtime. In contrast, this paper presents a static meta-programming facility that ensures that all derived rules are well-typed.

Evita Raced [10] adds a metacompiler to NDlog – an NDlog compiler written in NDlog. Their focus is on enabling extensible and reconfigurable compiler optimizations rather than security.

10. CONCLUSION

We present the design and implementation of SecureBlox, a declarative system that unifies a distributed query processor with a security policy framework. SecureBlox extends *LogicBlox* – an emerging commercial Datalog-based platform for enterprise software systems – with distributed execution, and support for meta-programming through BloxGenerics.

We present a series of customizable security mechanisms enabled by SecureBlox, including various forms of authentication, encryption, and delegation. Using these basic security building blocks, we demonstrate the use of SecureBlox to support a range of secure applications, including a path-vector routing protocol and a parallel hash-join operation. Our evaluation results demonstrate SecureBlox's ability to specify and implement secure distributed systems using a range of cryptographic schemes that achieve trade-offs in performance and security.

As our ongoing research, we are exploring various practical deployment scenarios for SecureBlox, including secure Internet communication, and multi-user federated cloud computing environments that require secure data sharing and integration among users. We are also actively exploring the use of the declarative framework for formally reasoning about the security properties of deployed programs written in SecureBlox.

11. ACKNOWLEDGMENTS

We would like to thank Greg Brooks, Soeren Olesen, Wael Sinno, and David Zook at LogicBlox for their instrumental support in building SecureBlox and for helpful discussions about language design. This work was supported in part by Air Force Office of Scientific Research Grant No. FA95500810352, as well as NSF Grant Nos. IIS-0812270, CNS-0831376, and CAREER CNS-0845552.

12. REFERENCES

[1] Cassandra: Distributed Access Control Policies with Tunable Expressiveness. In *Proceedings of the Fifth IEEE International Workshop on Policies for Distributed Systems and Networks (POLICY)*, page 159. IEEE Computer Society, 2004.

[2] M. Abadi. On Access Control, Data Integration, and Their Languages. In *Computer Systems: Theory, Technology and Applications, A Tribute to Roger Needham*, pages 9–14. Springer-Verlag, 2004.

[3] M. Abadi and B. T. Loo. Towards a Declarative Language and System for Secure Networking. In *Proceedings of the Third USENIX International Workshop on Networking Meets Databases (NetDB)*, pages 1–6. USENIX Association, 2007.

[4] P. Alvaro, T. Condie, N. Conway, K. Elmeleegy, J. M. Hellerstein, and R. C. Sears. BOOM Analytics: Exploring Data-Centric, Declarative Programming for the Cloud. In *EuroSys*, 2010.

[5] P. Alvaro, T. Condie, N. Conway, J. M. Hellerstein, and R. C. Sears. I Do Declare: Consensus in a Logic Language. In *Proceedings of the Fifth International Workshop on Networking Meets Databases (NetDB)*, 2009.

[6] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized Trust Management. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy (SP)*, page 164. IEEE Computer Society, 1996.

[7] M. Bravenboer and Y. Smaragdakis. Exception Analysis and Points-To Analysis: Better Together. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis (ISSTA)*, pages 1–12. ACM, 2009.

[8] M. Bravenboer and Y. Smaragdakis. Strictly Declarative Specification of Sophisticated Points-To Analyses. In *Proceeding of the Twenty-Fourth ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pages 243–262. ACM, 2009.

[9] W. F. Clocksin and C. S. Melish. *Programming in Prolog*. Springer-Verlag, 1987.

[10] T. Condie, D. Chu, J. M. Hellerstein, and P. Maniatis. Evita Raced: Metacompilation for Declarative Networks. *Proceedings of the VLDB Endowment (PVLDB)*, 1(1):1153–1165, 2008.

[11] J. DeTreville. Binder, a Logic-Based Security Language. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy (SP)*, page 105. IEEE Computer Society, 2002.

[12] A. Deutsch, A. Nash, and J. Remmel. The Chase Revisited. In *Proceedings of the Twenty-Seventh ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pages 149–158. ACM, 2008.

[13] R. Dingledine, N. Mathewson, and P. Syverson. Tor: The Second-Generation Onion Router. In *Proceedings of the Thirteenth conference on USENIX Security Symposium (SSYM)*, pages 21–21. USENIX Association, 2004.

[14] R. Geambasu, S. Gribble, and H. M. Levy. CloudViews: Communal Data Sharing in Public Clouds. In *Workshop on Hot Topics in Cloud Computing (HotCloud)*, 2009.

[15] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 157–166. ACM, 1993.

[16] R. Huebsch, J. M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proceedings of the Twenty-Ninth International Conference on Very Large Data Bases (VLDB)*, pages 321–332. VLDB Endowment, 2003.

[17] T. Jim. SD3: A Trust Management System with Certified Evaluation. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy (SP)*, page 106. IEEE Computer Society, 2001.

[18] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in Distributed Systems: Theory and Practice. *ACM Transactions on Computer Systems (TOCS)*, 10(4):265–310, 1992.

[19] N. Li, B. N. Grosof, and J. Feigenbaum. Delegation Logic: A Logic-Based Approach to Distributed Authorization. *ACM Transactions on Information and System Security*, 6(1):128–171, 2003.

[20] LogicBlox Inc. http://www.logicblox.com/.

[21] B. T. Loo, T. Condie, M. Garofalakis, D. E. Gay, J. M. Hellerstein, P. Maniatis, R. Ramakrishnan, T. Roscoe, and I. Stoica. Declarative Networking: Language, Execution and Optimization. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 97–108. ACM, 2006.

[22] B. T. Loo, T. Condie, J. M. Hellerstein, P. Maniatis, T. Roscoe, and I. Stoica. Implementing Declarative Overlays. In *Proceedings of the Twentieth ACM Symposium on Operating Systems Principles (SOSP)*, pages 75–90. ACM, 2005.

[23] B. T. Loo, J. M. Hellerstein, I. Stoica, and R. Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 289–300. ACM, 2005.

[24] W. R. Marczak, D. Zook, W. Zhou, M. Aref, and B. T. Loo. Declarative Reconfigurable Trust Management. In *Fourth Biennial Conference on Innovative Data Systems Research (CIDR)*, 2009.

[25] M. Meier, M. Schmidt, and G. Lausen. On Chase Termination Beyond Stratification. *Proceedings of the VLDB Endowment (PVLDB)*, 2(1):970–981, 2009.

[26] Predictix. http://www.predictix.com/.

[27] Semmle. http://semmle.com/.

[28] Who uses Tor? https://www.torproject.org/torusers.html.en.

[29] W. Zhou, Y. Mao, B. T. Loo, and M. Abadi. Unified Declarative Platform for Secure Networked Information Systems. In *Proceedings of the 2009 IEEE International Conference on Data Engineering (ICDE)*, pages 150–161. IEEE Computer Society, 2009.