Secure Authorization for Federated Environments (SAFE) Overview and Progress Report [1]

Jeff Chase and Vamsidhar Thummala

Technical Report CS-2014-003
Department of Computer Science
Duke University
June 2014
revised 9/3/14

1 Introduction

SAFE is a research project in logic-based authorization and trust management for federated systems and networks, focusing on networked cloud infrastructure services and software-defined networking. We also use the name SAFE for software built in the project. The name "SAFE" is an acronym for Secure Authorization for Federated Environments.

SAFE is an example of declarative trust management. Principals use a declarative language to make statements about one another and about objects in the system. These statements are secure assertions: they are authenticated and the source (speaker) of every statement is tracked. Principals reason from these statements according to policy rules, which are also written in the declarative language. Principals run local copies of an inference engine that interprets the language to make trust decisions from local beliefs, local policy rules, and statements (e.g., credentials) received from other participants. One canonical example of a declarative trust management system is SPKI/SDSI [11].

In SAFE, the declarative language is a logic formalism—a trust logic. In this respect SAFE borrows heavily from previous works variously known as credentials-based authorization, logic-based authorization, and role-based trust management. At the core of SAFE is a standard logic based on pure datalog augmented with the classic **says** operator of BAN belief logic [15] and limited constraint domain features of Role-based Trust [16]. (See the Appendix for a summary of trust logic concepts and terminology.)

The SAFE project builds on the earlier research in logic-based trust management by focusing on logical trust as a systems problem. Elements of the SAFE project include integration with application service frameworks, a deployment structure that facilitates cross-language interoperability, programming tools to construct policies and credentials for logical trust, and a decoupling from the external representations to transport the logic. Like earlier systems for logical trust, SAFE exports and shares logic material as digitally signed bundles—certificates. Therefore, it is also crucial to address common challenges for certificate management: storing, sharing, organizing, discovering, caching, revoking, reissuing, and refreshing certificates.

2 SAFE in Context

We are developing SAFE as an enabling tool for the Silver project (`silver.cs.unc.edu`). Silver is an NSF *Frontier in Secure and Trustworthy Cyberspace* project focusing on new security architectures for cloud computing, incorporating software-defined access control for cloud tenants, attestation of hosted services and their security properties by cloud providers, secure cross-tenant data sharing (e.g., confidentiality-preserving

[1]This paper is based upon work supported by the US National Science Foundation through the GENI Initiative and under NSF grants OCI-1032873, CNS-0910653, and CNS-1330659, and by the State of North Carolina through RENCI.

data mining), secure software-defined networking, and cloud federation. These elements require a flexible means to specify access control policies and convey trust information from one party to another securely. A common theme is to expose security properties of programs, providers, resources, and tenants, and use them as a basis for automated reasoning about trust and control within an open-market cloud service ecosystem.

To this end, Silver envisions a declarative security language as a building block. The language must be suitable to capture various security properties in a rigorous way, and reason about them at runtime according to declarative policy. The general framework is familiar from many previous systems. Applications incorporate a standard authorization policy checking agent (a guard), which checks access control policies specified in an interpreted policy language. The guard may also accept assertions and policy rules from other parties, according to its installed policies. Within this broad framework we are considering language alternatives balancing expressiveness and simplicity, with the goal of creating one or more prototypes that are easy to use, practical, and fast.

SAFE is one point in this continuum, building on the model of authorization logics. In the typical model the requester is a client who requests access from a server; the server is the authorizer. We note at the outset that although we use trust logic for client access control in this way, Silver emphasizes statements about services—cloud providers and their tenants—for use by their clients. That is, the access-control view is "flipped around" so that the clients may also act as authorizers. In this respect our approach extends the conventional identity-based Web security architecture—in which the client examines credentials that bind server SSL keys to DNS domain names—to incorporate rich statements about services and their security attributes, independent of their names.

While placing clients in the role of authorizers does not impact the logic itself, it does have other implications. For example, support for logical attestation [19, 20] is crucial, so that a host may make statements about the program/image that a tenant instance is running. Also, confidentiality of credentials is a secondary concern to the extent that these credentials are advertisements by services (providers) to their potential customers. The requests are initiated by the authorizers, so we can exclude probing attacks [12].

SAFE is an outgrowth of earlier research on declarative trust for federated community clouds, funded from the NSF GENI project and NSF's Trustworthy Computing program (CNS-0910653). At the start of the Silver project we had a working prototype implemented in Scala, a roadmap to advance the prototype, and several "worked examples" including the GENI authorization system and Trusted Platform-as-a-Service [5]. During the first year of the Silver project we advanced the SAFE language, system, and implementation in various ways as summarized in this paper. A companion paper presents the GENI example with the current version of SAFE in detail [9].

3 SAFE Logic: Slog

We conceived the SAFE logic ("slog") as a "minimalist" trust logic with the features we need and nothing more. It is not our intent to advance the state of the art in trust logic, but rather to overcome practical barriers that inhibit its adoption as a rigorous and powerful tool for building secure systems. A dominant design goal for slog is to keep the inference engine simple enough to embed it in real systems built using various programming language environments.

Slog is based on pure datalog. Datalog is a well-known declarative logic language with a standard syntax and well-understood formal properties. Our premise is that datalog is the maximally expressive logic language that is provably tractable. Slog is similar in spirit to Binder [10] and SD3 [14]; more recently SENDLog [3] also takes this approach. Slog is positive, monotonic, and constructive. While at present we see little need for threshold/manifold structures or negation, slog could grow to incorporate them if we decide that they are useful, following SecPAL [4] and NAL [19].

Slog augments datalog with the **says** operator. Every predicate and every atom has by convention a first parameter representing a principal who says it (the *speaker*). This simple addition makes slog a belief logic in

which shared conventions for predicate names are needed only for interoperability, but not for soundness. Consider the following example statement:

```
p(X) :- alice:p(X), bob:p(X).
```

These statement reads *"self infers p(X), for any given X, if Alice says p(X) is true and Bob says p(X) is true"*. The `:-` is datalog/prolog syntax for logical implication: this statement is a *rule*. The text to the left of the `:-` is the *head* of the rule, and the text to the right is the *body*. The head is a single atom—a factual atomic statement consisting of a logical predicate (`p`) applied to specific parameters, which may include one or more variables (`X`). A rule allows the interpreter to infer that the head is true, for some substitution of its variables with constants, if the body is true under that substitution. The body is a sequence of atoms (called *goals*) separated by commas, which indicate conjunction: all of the atoms in the body must be true for the rule to "fire". All variables in the head must also appear in the body.

In this example each goal atom names a specific speaker: the atoms in the body are prefixed with a **says** operator (`:`) naming the principals `alice` and `bob`. The head does not specify a speaker: by convention, if an atom does not specify a speaker then the default speaker is `$Self`—the local authorizer who applies the statement. We may view this rule as a trust policy that leads an authorizer to believe `p(cindy)` if both `alice` and `bob` agree that it is true.

Not every statement is a rule with a head and a body. Consider these statements:

```
alice:p(cindy).
bob:  p(cindy).
```

These statements are assertions of fact: `alice` and `bob` say that the predicate `p` is true of `cindy`. A fact is essentially a degenerate rule with no body—and therefore no variables. In general, an authorizer accepts such an assertion (or any statement) as a basis for reasoning only if the statement is authenticated by some cryptographic means, so that the authorizer can be sure that `alice` and `bob` really did say `p(cindy)`.

It is another question whether the authorizer itself believes that `p(cindy)` is true. But if the authorizer has the rule above installed as a local policy rule, then it does believe `p(cindy)`, once it has received the authenticated statements from `alice` and `bob`. Of course, all three principals must agree on the real-world meaning of the predicate `p` for the inference to be sound. But if the authorizer receives a statement spoken by another principal (say `mallory`), it simply ignores the statement if there is no other policy rule delegating power to `mallory` to influence the authorizer's beliefs. In that case it does not matter what `mallory` means by `p` or if her statement is a lie.

In this way, the rules in a trust logic specify whose assertions are accepted as a basis for any given inference. Slog rules may quantify over the speaker. Consider the rule:

```
p(X) :- Y:p(X), q(Y).
```

This rule reads *"self infers p(X), for any given X, if some principal Y says p(X) is true, and self believes that q(Y) is true"*. This rule states that the authorizer believes an assertion matching `p(X)` if it is spoken by any principal possessing a certain attribute or meeting a certain condition (`q(Y)`). Other rules may specify how the authorizer can infer that the condition is true. This kind of statement is called an *attribute-based delegation* [18].

Our earlier work with GENI used a different logic formalism called Role-based Trust [18] or Attribute-Based Access Control (ABAC). RT/ABAC has multiple variants of different strengths. None are more powerful than datalog: in fact, they can transform syntactically to datalog (or to slog). An open-source RT/ABAC toolkit from DETER/ISI implements the RT0 subset, which is less powerful than datalog and equivalent to SPKI/SDSI with conjunctions [17]. We determined that RT0 was adequate for the initial core of the GENI

authorization system [7, 8], but it is not sufficiently powerful to meet our other goals. For example, RT0 cannot represent statements about objects that are not principals, such as programs or images. Further investigation suggested that the more powerful RT1 and RT2 offered no substantive advantage over pure datalog. While RT1 and RT2 support tractable constraint domains [16], these are straightforward to add to datalog. Ultimately we decided (in spring 2013) to build our own engine for standard datalog with minimal trust logic extensions (e.g., **says**). That engine is the core of SAFE.

4 Elements of the SAFE System

The GENI experience exposed various practical obstacles to leveraging trust logic in complex networked systems. In particular, there is a need for tools to generate and import/export authenticated credentials (e.g., as certificates), some of which may naturally contain multiple logic statements, and to otherwise manage sets of credentials, the flow of credentials through the system, and assembly of queries and context sets for access checking. The SAFE system addresses these challenges.

What is novel about SAFE is the integration of the trust logic with a scripting language ("slang") and with SafeSets, a decentralized system for sharing logic content based on a standard open-source key-value store (currently Riak). These elements work together to simplify and automate many aspects of networked trust.

Slang and SafeSets are organized around the abstraction of *logic sets* — sets of slog statements. Logic sets are first-class objects named by secure references called *tokens*. Slang offers programming constructs to build and modify logic sets, link them to form unions, post them to SafeSets, pass them by reference, and add them to query contexts for trust decisions. A posted set is accessible to any client that knows its token, but only its issuer can modify it.

SAFE runs as an interpreter process with one or more slang programs loaded into it. The interpreter runs on a JVM within its own process. It exposes a REST API to the application (the authorizer) over a protected socket. Applications may call this API directly, or indirectly through hooks added to a programming framework. For example, consider a server that uses SAFE to apply access control checks for incoming requests, depicted in Figure 1. The server checks a request by issuing a logic query (a *guard* predicate) against a logic set containing credentials and policy rules (the *context* for the query). A server framework can assist by querying the guard associated with each requested method, passing standard parameters from the request, and caching the result. These parameters include the subject and object of the request and a token referencing the caller's credentials (`$BearerRef`), which may be stored in SafeSets. Section 4.3 discusses this example in more detail.

The REST API enables the application to load (and reload) slang code into the interpreter and invoke it at named entry points defined in the code. As the slang program executes, the SAFE process acts as a client of SafeSets to post and fetch logic sets. It may also fetch URLs or access other Internet services as directed by the program, but it presents no service interface to the external network. These choices assure that SAFE is portable and interoperable. They also enhance security for server applications that use SAFE: the application's signing keypair (`$SelfID`) is isolated from its Web-facing front end.

SAFE is implemented with an extensible structure in about 4000 lines of Scala, including the interpreter (1800 lines), SafeSets (1000 lines), and crypto support (700 lines).

4.1 Slang: A Logic-Based Scripting Language

The slang scripting language runs on the same interpreter core as slog. Slang adds limited functional features to construct logic sets and keying material and to manipulate them as values. It also supports environment variables and substitution. These features are useful for generating credentials and policy rules from templates defined directly in the code.

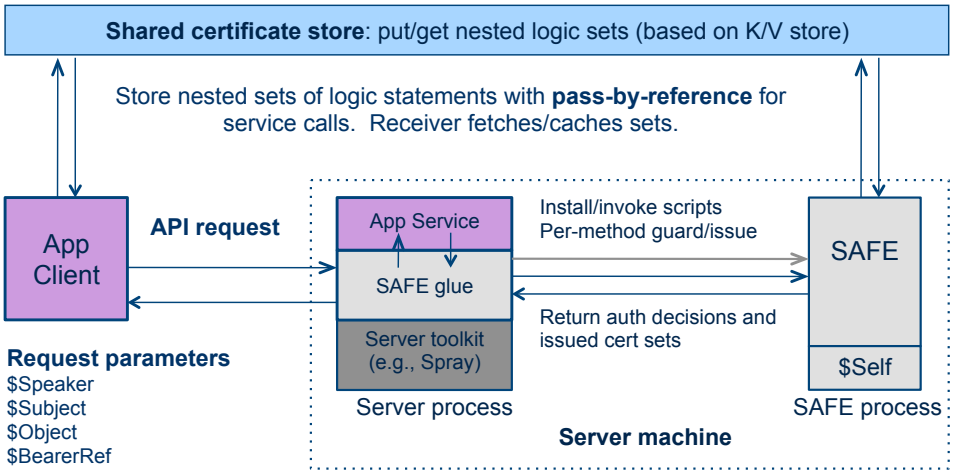

Figure 1: Server access control using SAFE. The SAFE interpreter may run as a separate process with a copy of the interpreter for the SAFE languages—slog and slang—and the principal's signing key (`$SelfID`). The server application installs slang code in the SAFE process to configure the access control policy and process user credentials for access checks. Credentials are passed as references to signed logic sets in a network certificate store (SafeSets). The SAFE process fetches logic sets on demand, validates the signature and speaker, and caches them for use by the logic interpreter.

Slang is a hybrid functional-logic programming environment using an extended logic syntax. A slang program is a set of logic rules similar to the slog rules shown above. A crucial difference is that slang rules may act as functions that return values, including logic sets. More generally, slang provides a library of builtin functional predicates for string manipulation, crypto, networking (URL fetch), and other primitives.

Functions are not permitted in pure datalog because inference involving functions is intractable in general. In contrast to datalog (and slog), slang defines a deterministic procedural semantics for evaluation. The slang interpreter evaluates the goals of a rule from left to right: if the last goal of a rule evaluates to a value, then that value is returned as the result of the rule. Thus any slang rule can act as a function. In general, slang programs execute as a functional evaluation rather than inference: the evaluation follows a deterministic path with no backtracking. This behavior follows from a convention that for each slang predicate there is at most one rule with a matching head.

Like many scripting languages, slang is a string-oriented language with "duck typing". All values are strings. Slang includes various builtin functions to manipulate strings in particular formats and interpret them in a particular way. Examples include strings that encode structured data (such as lists or dates) or names within the SAFE system (e.g., as a reference to a logic set). Any string in the expected format is legal as a parameter.

A slang rule is tagged with a `def` keyword that declares a type for the rule. The various slang rule types have additional behaviors to integrate with the application and with SafeSets, and to extend the scripting primitives. A rule tag is one of the reserved keywords `defcon`, `defguard`, `defun`, `defenv`, `defpost`, or `definit`. The next few subsections summarize these rule tags and illustrate with simple examples.

Refer to the companion paper [9] for an extended discussion of how to use slang to implement a complex networked system: the GENI authorization system. The GENI example includes global objects (slices) with hybrid capability-based access control, multiple object authorities, authority services for identity and project membership, a declarative federation trust structure, and dynamic policies with policy mobility.

4.1.1 `defenv and definit`

Slang supports *environment variables*. Environment variables are similar to logic variables: they have similar naming conventions, and they may appear in facts and rules. The difference is that an environment variable is always bound to a value before evaluation of any statement in which it appears. Thus environment variables appearing in a statement are substituted with constant values before evaluation: their values do not change during evaluation. Datalog variables are scoped to a single rule, while environment variables are scoped to the entire program. Environment variable names are prefixed with $ to distinguish them from regular logic variables.

A `defenv` rule assigns a value to an environment variable of the same name. If the variable is referenced before a value is assigned to it, then the corresponding `defenv` is evaluated and the result is assigned to the variable. These rules are useful to initialize parameters in a program. Here is an example that initializes the `$SelfID` environment variable, which is designated to hold the local principal's keypair.

```
defenv SelfID() :-
   spec("Load a key pair from an external file"),
   getKeyPair(@("/home/user/keypair.pem")).
```

SAFE uses `$SelfID` to sign exported logic sets and generate secure tokens. A slang program must initialize `$SelfID` to a registered keypair before posting any content. This example reads the keypair from a local pem file. The builtin `@` operator returns the contents of a document given a location, which is typed by its format and extension.

A related slang rule time is `definit`. Each `definit` rule is evaluated exactly once when a slang module is loaded. These rules are useful for other initialization tasks, such as minting a keypair or posting identity credentials.

4.1.2 `defcon`

A `defcon` rule is a function that creates or modifies a labeled logic set and returns its value—a *set constructor*. Consider this example:

```
defcon policySnippet() :-
 {
     p(X) :- Y:p(X), q(Y).
     label("policy rule").
 }.
```

Slang introduces {} notation to specify sets. The result of evaluating a {} goal is a string reference to a logic set with contents listed in the brackets. Each item listed within the brackets is a slog statement. The constructor returns a logic set containing the listed statements. A statement in a logic set may be a fact using a pre-defined builtin *meta-predicate* (e.g., `label`). The meta-predicate facts direct the set's encoding as a certificate or enable linkages with other sets.

In this example the constructed logic set has a label and a single slog policy rule—the example rule discussed above. In a `defcon` the `label` meta-predicate assigns a string label for the constructed set. The label must be unique among all locally constructed sets: if the label value refers to an existing set, then the `defcon` modifies it rather than constructing a new set. The label parameter may be a template: it may include environment variables whose values are substituted at runtime to generate the label value. Another set constructor may use the `link` meta-predicate to incorporate this set by reference, naming it by its label.

```
defcon mypolicy() :-
```

```
RuleRef := label("policy rule"),
{
    q(alice).
    link(RuleRef).
    label("my policy").
}.
```

This constructor assembles a new policy set that includes the previous `policy rule` set as a subset. It adds a new slog fact `q(alice)`. The new fact combines with the policy rule to complete the hypothetical policy to decide who to believe about assertions of `p(X)`: an authorizer using this policy believes only `alice`.

This example uses the `link` meta-predicate to link to the `policy rule` set, referenced by its string label. A SAFE instance has a single global label space for locally constructed sets. The first goal of this rule introduces a shorthand syntax for a slang or slog atom to assign a value to a variable (`Ref`) and return its value. The assignment uses the builtin `label` predicate as a function: used in this way, `label` returns a token for the local set named by the given label string. The `link` meta-predicate receives this token as a parameter.

4.1.3 `defguard`

The `defguard` rules define external entry points to the slang program for access checking of incoming requests. Each access-checked operation has a corresponding `defguard` rule in the slang program. The rule defines the guard query for that operation and the context for the query. Consider this example:

```
defguard mymethod() :-
 PolicyRef := label("my policy"),
 {
    link(PolicyRef).
    link($BearerRef).
    p($Subject)?
 }.
```

This guard rule defines an access policy for a particular operation—by convention the operation with the same name (`mymethod`). The last goal of the guard rule uses {} notation to define a logic set, which is used as the context for the authorization query. The guard code uses `link` to specify any subsets to include in the context, in the same fashion as the `defcon` example above. The context includes the policy set from the `defcon` example, named by the label. It also includes the query statement itself (terminated by a `?`). The guard dispatches the context to the slog inference engine, which executes the query.

The `defguard` rules are the key to how SAFE integrates with applications. Section 4.3 outlines our prototype for integration with a web server framework, including conventions to define `$BearerRef` and `$Subject` as special per-request environment variables. The value of `$Subject` is the authenticated name of the principal (e.g., a client) who issued the request. The value of `$BearerRef` is a token referencing the requester's credentials to support the request. (Refer also to Figure 1.)

The guard example presumes that the application is a service following those conventions. The `link` statement fetches the logic set referenced by the `$BearerRef` token and includes those credentials in the query context. For example, suppose the requester's credential set includes a valid assertion from `alice` that the requester has property `p`. Then the guard query evaluates to **true** and the request is allowed.

4.1.4 `defpost`

The `defpost` rules post one or more sets to SafeSets, generating secure tokens for those sets as needed (see below). The `defpost` rules are useful to issue credentials or to post policies for use by other principals. Consider this example:

```
defpost postPolicy() :-
  [mypolicy()].
```

A `defpost` rule posts the value of its last atom, which is a list of references to locally constructed logic sets. In any slang rule a goal that uses `[]` notation returns the specified list; this is similar to the use of {} notation to specify a set, as in the `defcon` and `defguard` examples above. In this example the list has a single element: a reference to the policy set constructed above.

When a program posts a logic set, the slang runtime generates a globally unique and secure set token from the set's assigned label. (See below.) The system materializes the logic set as a certificate that is digitally signed under the issuer's keypair (`$SelfID`). In general, the speaker of every posted statement is the issuer's principal name (`$Self`). SAFE does offer support for third-party attributions (**speaksFor**) at the slang level, but we do not discuss it here.

SAFE supports compact, reliable encoding of logic sets in X.509 certificates (using a string encoding within an attribute field) and also in a native SAFE format. The crypto layer represents all semantic content of any signed certificate internally in common logic, including builtin predicates for meta-attributes such as expiration time, encoding type, and so on. It generates the encoded cert from a logic set containing the required meta-attributes as facts, which are easy to specify directly in slang set constructors. The native SAFE cert format is not subject to the arbitrary length constraints of x.509 certificates, and also improves compactness by hashing public keys embedded as principal names in the logic.

4.1.5 `defun`

The `defun` tag marks a generic functional rule. Slang rules tagged with `defun` may include embedded native Scala code, which is compiled on-the-fly during load (or reload). This feature is useful to extend the available builtin functions for slang scripting.

SAFE includes a library of `defun` builtins to interface to standard crypto and networking functions available in Scala, Java, and other JVM languages. For example, slang programs can bootstrap trust from the existing Internet by fetching public keys of their peers via secure web URLs.

4.2 Sharing Logic Sets

Slang and SafeSets offer an integrated solution for sharing authenticated logic sets in a networked system. Slang programs are isolated from the details of the SafeSets API: the slang runtime handles secure token generation, post, fetch, and cryptographic operations transparently.

The slang layer defines conventions for secure names used in logic sets, to ensure that they are unique and authenticated. A principal is named by the hash of its public key. An object is named by a GUID qualified by the name of its owning principal, who **controls** the object. These are slang conventions: slog requires only that the names are unique and distinct.

Every logic set has a unique token that is cryptographically secure. To derive the token SAFE concatenates the issuer's public key hash (`$Self`) with the label value (which is locally unique) to form a secure global name, then hashes it to a fixed-width token. Currently a token is a 256-bit SHA hash encoded as a 44-byte base64 string.

Any slang program that knows a token can fetch a logic set from SafeSets. More generally, anyone who knows the label and the owning principal can synthesize the token and fetch the set. Post requests on SafeSets are signed under the `$SelfID` keypair used to generate the token. The SafeSets service verifies cryptographically that the requester's signature is valid and that its public key is hashed into the set token (the requester must supply the label). These checks ensure that only the rightful owner of a token may post to it. In this sense SafeSets tokens are secure.

SAFE fetches logic sets automatically on first use. After fetch, SAFE validates the signature to authenticate the stated speaker of each imported statement. If the certificate is valid, its contents are extracted into an in-memory logic set, including meta-attributes. Sets created in slang or imported from SafeSets are cached in an in-memory *set cache*. The certificates are also cached on local disk.

As shown above, set constructors may include other sets by reference using the builtin `link` predicate. Set linking enables the construction of logic sets that link to all credentials needed for a given class of access control decisions. Internally, each set is represented as a DAG whose leaves are certificates. These DAGs are cached incrementally. A leaf subset expires from the cache at the expiration time of its containing certificate. If an expired certificate is reissued, then a fetch pulls the fresh certificate from the store automatically.

Set linking is also useful to define contexts for authorization queries in `defguard` rules. Different trust decisions by the same principal may reason from different contexts. A core design goal of SAFE is to manage contexts carefully to reduce inference cost by limiting their size, tailoring each context to include only the information relevant to each guard query. For example, an object server might use a common set of policy rules with separate per-object sets representing logic specific to each object; a request for any given object validates against a context containing only the rules for that object, and none other. To support this goal, the inference engine assembles the context for a slog query from multiple subsets, each referenced by a `link` statement in the guard.

When a context is assembled and dispatched to the inference engine, SAFE *renders* it to an internal context format and caches it in a *context cache*. A rendered context set is indexed to speed up the inference engine, which must search for rules with heads matching each goal. We have enhanced the context cache and inference engine to assemble a context from multiple subcontexts in the context cache (e.g., subcontexts containing policy rules and facts pertaining to a given subject and object). We also added support to invalidate a cached context at the earliest expiration time of any statement it contains.

4.3 Server Integration

As depicted in Figure 1, application servers can use SAFE to check access control for client operations on the objects they serve. In general, server applications are built using general-purpose service frameworks that handle the details of network communication, URL parsing, request dispatch, and threading. The service application itself consists of an implementation of the service API: it is a set of methods that plug into the server framework.

Suppose that each API method of the service has a corresponding guard in the slang program. When a request enters the Web application or service framework, it invokes SAFE to evaluate a declared guard whose name matches the requested method, passing a list of variables named in the request. SAFE evaluates the guard and returns the result to the service framework, which rejects the service request if the result is false. Ideally there is no change to the application itself, other than defining slang guards for each method.

In our experimental work we focus initially on the Akka framework, a server and runtime toolkit for highly concurrent REST services. We added SAFE "glue" code added to the framework to gather parameters for the request and invoke the corresponding guard entry point before invoking the application method for a request. The parameters are passed into the slang program as named environment variables.

This section summarizes the conventions for web service integration in our prototype, which builds on the secure naming conventions for slang and SafeSets outlined above. Several environment variables with predefined names are always present when the guard is invoked.

- `$Subject` contains the authenticated principal name of the requester. It is the responsibility of the framework to authenticate the requester: one approach is to use secure HTTP as the transport and accept a client certificate.
- `$BearerRef` contains a single token passed by the requester, representing the credentials to support the request.

- `$Object` is the identifier of the target object, if the request has a target object.

The guard may reference these special environment variables. They differ from other environment variables in that their values are local to a request to SAFE's API. Each request executes on a separate thread with its own binding for these variables.

We enhanced integration with Akka by adding initial support for a query result cache on the Web server side. The result cache optimizes repeated operations by a given subject on a given object.

5 Issues for Credential Management

The SAFE approach to managing credentials is simple and powerful, and addresses a number of longstanding challenges for credential management.

Renewal. An issuer may renew an expired certificate by posting the renewed certificate to SafeSets with the same token, overwriting the expired certificate. If a SAFE authorizer encounters an expired set it uses the token to fetch a new version automatically, and retries the query.

Revocation. An issuer may change a posted logic set at any time or "poison" the set to invalidate it. Of course, a change or poison does not take effect if an authorizer uses an old copy of the set from its cache. An authorizer may refresh sets in its cache at its discretion, even if they have not expired. An issuer may control the expiration times to bound the time that a set remains in an authorizer's cache.

Rotation. SAFE GENI names principals by their public key hashes. If a principal loses or rotates its keypair, then sets that incorporate the stale key in their tokens must be regenerated. Note that these changes must propagate backwards one hop in the DAG: if any set links to a stale set, then an owner must update the link. For example, if an issuer loses its key, then all subjects it issues credentials to must link their subject sets to a new issuer set. We have planned features for SAFE to simplify key rotation, but these are not yet implemented.

The SAFE approach to managing credentials also raises some potential concerns.

Malicious content. Issuers may write malformed certificates to SafeSets or generate a malformed credential DAG, e.g., by creating cycles in the DAG. The SAFE fetch procedure rejects malformed certificates and detects cycles. Valid certificates contain only slog statements, which share the termination properties of pure datalog: all queries terminate. However, issuers may create very large or costly logic sets to mount a denial of service attack. An authorizer may bound the size of incoming logic sets and query contexts at its discretion.

Accountability. Policy mobility relies on participants to enforce the policy rules of others. In general, entities control their own authorization decisions and have power to do harm only to the extent that others trust them. For example, a GENI aggregate that ignores policy conditions may be unduly promiscuous with its own resources, but it cannot affect access to the resources of others. Moreover, all entities are strongly accountable (in the sense of CATS [22]) for credentials they post to SafeSets representing the result of access decisions. These credentials are digitally signed and non-repudiable, and we could arrange for a third-party auditor to check the proofs. Accountability is an active research topic.

Confidentiality. Synthesized tokens raise the issue of confidentiality of policy rules and other logic material stored in SafeSets. We presume that a principal's subject set is public to anyone who knows the principal's public key or its hash. If an entity wants to protect a confidential logic set it may salt the label with another hash. If a label has a hash within it, then an attacker must obtain the hash value by some means in order to access the set: it is infeasible to guess a token that is effectively random. We emphasize that the protection for writing to SafeSets is stronger: a client must possess a principal's private key in order to write to a logic set that the principal controls.

Reclamation. Logic material may accumulate in SafeSets over time. SafeSets may delete any set after it has expired: all logic sets have expiration times. Even so, issuers may use unreasonable expiration times or

simply post useless data to the store. SafeSets authenticates each issuer by its public key, but quotas are of no help if an issuer can mint new keys at will. One option is to apply a SAFE access check for posting. Another option is to arrange the store so that each issuer provides and manages its own storage (e.g., via a Web server).

SafeSets failure. Managing SafeSets as a decentralized key-value store can be a scalable and reliable solution. One or more entities may control the SafeSets servers. A faulty or malicious server can destroy content or block access to it. However, it cannot subvert the integrity of the system because all logic sets are signed by their issuers.

Appendix: Logical Trust

A *logic* is a language for forming declarative boolean assertions called *statements*. Statements include *facts* and *rules*. A useful logic includes a semantics that defines whether a fact is true or false, and a syntactic inference procedure that applies rules to derive true facts from other true facts. The procedure is implemented in a piece of software called an interpreter or inference engine.

A trust logic is a logic with features for reasoning about statements and beliefs of multiple principals in a distributed system. Entities in real systems may use trust logic to make trust decisions, based on local policy rules and authenticated statements made by other entities and exchanged over the network. Security policies are specified rigorously in logic, so they are easier to verify and change.

5.1 Datalog

Datalog is a simple and accessible logic system. We focus on "pure" datalog, also called safe ordinary datalog. Pure datalog has various constraints on the use of variables, and it has no functions or negation. Pure datalog is a positive, monotonic, constructive logic. This just means that it cannot prove that a statement is false, and acquiring new knowledge does not invalidate what is already known. Datalog is tractable: any problem expressible in datalog is polynomial [13, 21].

Datalog statements are built up from atomic statements based on a vocabulary of *predicate* symbols, which are boolean relation constants. Each predicate symbol has a signature of some arity n: the predicate is defined over n parameters. An *atom* is an n-ary predicate symbol applied to n terms: $P(t_0, ..., t_n)$. A *term* is either an unbound variable or it evaluates to a string constant that names an element in the universe (e.g., a principal or object, or an arithmetic value). An atom is *ground* if all of its terms are constants.

In pure datalog a *fact* is a single atom. All datalog facts are ground. A ground fact is either true or not, in any given universe. In practical use the truth of a statement corresponds to truth in the real world. However, the reasoning and inference within the logic system is purely syntactic. The real-world meaning of any non-reserved predicate symbol or term constant is a convention among the users of the logic.

A *rule* is a statement that enables logical inference by implication (modus ponens). Each rule has one head (left-hand side) and one body (right-hand side). The *head* is a single atom; the *body* is a conjunction of atoms called *goals*, separated by commas. A rule is understood as a simple production rule (if-then) under logical inference: the head is true if all goals of the body are true.

Rules may contain variables. A *variable* is a named term whose value is not bound as a constant; the variable can bind at runtime to values ranging over the constants. We use the convention that any term symbol that begins with a capital letter or _ is a variable.

Logic variables are understood to be universally quantified: a rule is true for all valid assignments of its variables. A valid assignment of variables binds each variable in the statement to exactly one concrete element of the universe, such that each goal of the statement is a true ground fact under substitution of each bound variable with its value wherever it appears in the rule.

5.2 Programs and Proofs

A logic *program* is an initial set of statements that are given as true. Given a program, a datalog system generates proofs in response to *queries*. A query is a ground fact. If the fact is true, then datalog yields a proof.

A *proof* is an annotated DAG demonstrating an inference of a queried fact from a program. The nodes of the DAG are ground facts. The leaves of the DAG are ground facts given directly in the program. The DAG has a root, which is the proven fact. Each non-leaf node is inferred from its children by application of some rule from the program, together with an assignment of values to all variables in that rule; the node is the head of that rule after substituting the variables with their values.

Datalog implementations use a search procedure to discover proofs. Datalog is a declarative language: its semantics are independent of any particular procedure. A practical implementation involves backtracking: if the engine fails to find a proof with one candidate variable assignment, then it changes the values and tries again. Proofs are verifiable in linear time.

5.3 Extending Datalog

Researchers have considered many ways to extend pure datalog. For certain useful extensions the resulting language is still tractable. We can add negation, if the program is "stratified". We can add pairwise ordering relations over integers or equivalent domains (e.g., strings). Therefore, we can add inclusion and containment of integer or string ranges (e.g., IP addresses, pathnames). Further extensions may be tractable for a subset of programs that conform to various additional constraints (e.g., [6]).

We can extend datalog in a simple way for use as a trust logic. To use datalog as a trust logic, we must add a **says** operator. The new form of an atom is $s : P(t_0, ..., t_n)$. The s term evaluates to a principal, who is the atom's *speaker*. (RT/ABAC and some other logics use a dot syntax instead of a colon.) Datalog-with-says is equivalent in power to datalog: it is equivalent to adding one parameter to every predicate to represent the speaker of each atom.

5.4 Using Logic for Authorization

In logic-based authorization systems, a principal (the *authorizer*) establishes its trust in another principal (commonly called the *requester*) by using a logic to construct and/or verify a proof. Each authorizer runs a local copy of the interpreter, which is part of its trusted computing base.

In the typical model the authorizer is a server and the requester is a client who requests access from a server. But it is also important for clients to validate their trust in the services that they depend on. Conventional Internet security relies on PKI identity infrastructure to associate each server with a DNS name known to the client. Logical trust extends the trust architecture to incorporate rich statements about the parties and their security attributes.

A *guard condition* is the statement to be proved for some given trust decision: the authorizer queries the interpreter for the guard condition, and grants trust to the requester if and only if the query yields a proof of the guard condition. The local policies of the authorizer determine the guard condition.

A *context* is a set of logic statements considered for a proof. The statements in the context represent the authorizer's beliefs and policy rules, including statements about the subject and object of the request and the rules governing access. In logic-based systems the context is a logic program: the inference is similar to query execution in Prolog or other logic programming systems.

5.5 Attributing and Authenticating Statements

The statements in a context may be obtained from other parties over the network. For example, a requester may pass credentials to support a request. Authorizers must consider the source of a received statement in deciding whether to accept it. In a trust logic every atomic statement has an embedded attribution to a *speaker*, a principal who **says** the statement, i.e., who makes or believes the statement. Attribution of statements and beliefs is built into the inference engine and propagated across every inference.

Any statements obtained from another party over the network are authenticated by cryptographic means. For example, they may be transmitted over a secure connection or encoded in a document (a certificate) that is digitally signed under a keypair. The authenticated source of a statement is its *issuer*.

Typically the issuer of an authenticated statement is the same as its speaker. If the issuer is not the speaker, then the receiver rejects the statement unless it believes the issuer is authorized to issue the statement on behalf of the speaker (**speaksFor**).

The rules in a trust logic specify whose assertions are accepted as a basis for any given inference. This construct precisely captures the idea of *delegation*. For example, a rule might state that the authorizer believes matching statements if they are spoken by some designated principal, or by any principal meeting certain conditions (*attribute-based delegation* [18]).

Any participant in a distributed system may create an object, assign it a name, and issue statements about it. These statements might include policy rules about who can access the object, or a statement about a client on whose behalf the object was created. An authorizer should accept such statements only if it believes that the speaker has authority over the object and its name (the speaker **controls** the object). Therefore, it is crucial to protect the security of names, e.g., by qualifying names with the public key of the controlling authority.

The **says** and **controls** operators originate with BAN belief logic [15] and the ABLP access control calculus [2]. In general, trust logics apply common axioms of ABLP access control calculus. In particular, every entity **controls** its own beliefs. An entity may state an inference rule for its own beliefs: a rule with a tag "A **says**" in the head is valid only if it is an authenticated statement made by A. If A does make such a statement, then other entities may use the rule to infer A's beliefs based on a given set of ground facts. These axioms are known as *Hand-off* and *Bind* respectively [1]. They enable sound trust policies and mobility of policy rules.

References

[1] M. Abadi. Variations in access control logic. In *Proceedings of the 9th International Conference on Deontic Logic in Computer Science*, DEON '08, pages 96–109, 2008.

[2] M. Abadi, M. Burrows, B. Lampson, and G. Plotkin. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems*, 15(4):706–734, Sept. 1993.

[3] M. Abadi and B. T. Loo. Towards a declarative language and system for secure networking. In *Proceedings of the 3rd USENIX International Workshop on Networking Meets Databases*, NETDB'07, pages 2:1–2:6, Berkeley, CA, USA, 2007. USENIX Association.

[4] M. Y. Becker, C. Fournet, and A. D. Gordon. SecPAL: Design and semantics of a decentralized authorization language. *Journal of Computer Security*, 18(4):619–665, Dec. 2010.

[5] A. Brown and J. Chase. Trusted Platform-as-a-Service: A foundation for trustworthy cloud-hosted applications. In *3rd ACM Cloud Computing Security Workshop*, Oct. 2011.

[6] A. Calì, G. Gottlob, T. Lukasiewicz, and A. Pieris. Datalog+/-: A family of languages for ontology querying. In *Proceedings of the First International Conference on Datalog Reloaded*, Datalog'10, pages 351–368, Berlin, Heidelberg, 2011. Springer-Verlag.

[7] J. Chase. Authorization and Trust Structure in GENI: A Perspective on the Role of ABAC. http://www.cs.duke.edu/~chase/geni-abac.pdf, June 2011.

[8] J. Chase, P. Jaipuria, S. Schwab, and T. Faber. Managing identity and authorization for community clouds. http://www.cs.duke.edu/čhase/geni-trust.pdf, Sept. 2012.

[9] J. Chase and V. Thummala. A Guided Tour of SAFE GENI. Technical Report CS-2014-002, Department of Computer Science, Duke University, June 2014.

[10] J. DeTreville. Binder, A Logic-Based Security Language. In *IEEE Symposium on Security and Privacy*, pages 105–113. IEEE, May 2002.

[11] C. Ellison, B. Frantz, B. Lampson, R. Rivest, B. Thomas, and T. Ylonen. SPKI Certificate Theory. RFC 2693 (Experimental), September 1999.

[12] Y. Gurevich and I. Neeman. DKAL: Distributed-knowledge authorization language. In *Computer Security Foundations Symposium*, CSF'08, pages 149–162. IEEE, June 2008.

[13] N. Immerman. Relational queries computable in polynomial time (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 147–152, New York, NY, USA, 1982. ACM.

[14] T. Jim. SD3: A trust management system with certified evaluation. In *IEEE Symposium on Security and Privacy*, pages 106–115. IEEE, May 2001.

[15] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4):265–310, Nov. 1992.

[16] N. Li and J. C. Mitchell. Datalog with Constraints: A Foundation for Trust Management Languages. In *Proceedings of the 5th International Symposium on Practical Aspects of Declarative Languages*, PADL '03, pages 58–73, 2003.

[17] N. Li and J. C. Mitchell. Understanding SPKI/SDSI using first-order logic. *International Journal of Information Security*, 5(1):48–64, Jan. 2006.

[18] N. Li, J. C. Mitchell, and W. H. Winsborough. Design of a role-based trust-management framework. In *2002 IEEE Symposium on Security and Privacy*, pages 114–130, 2002.

[19] F. B. Schneider, K. Walsh, and E. G. Sirer. Nexus authorization logic (NAL): Design rationale and applications. *ACM Transactions on Information System Security (TISSEC)*, 14, May 2011.

[20] E. G. Sirer, W. de Bruijn, P. Reynolds, A. Shieh, K. Walsh, D. Williams, and F. B. Schneider. Logical attestation: an authorization architecture for trustworthy computing. In *23rd ACM Symposium on Operating Systems Principles*, pages 249–264, 2011.

[21] M. Y. Vardi. The complexity of relational query languages (extended abstract). In *Proceedings of the Fourteenth Annual ACM Symposium on Theory of Computing*, STOC '82, pages 137–146, New York, NY, USA, 1982. ACM.

[22] A. R. Yumerefendi and J. S. Chase. Strong Accountability for Network Storage. *ACM Transactions on Storage (Selected papers from the 2007 Symposium on File and Storage Technologies*, 3(3), October 2007.