SAFE: A Declarative Trust Management System with Linked Credentials

Vamsi Thummala, Jeff Chase
Duke University
{vamsi, chase}@cs.duke.edu

25th Sep 2015

Abstract

We present SAFE, an integrated system for managing trust using a logic-based declarative language. Logical trust systems authorize each request by constructing a proof from a context—a set of authenticated logic statements representing credentials and policies issued by various principals in a networked system.

A key barrier to practical use of logical trust systems is the problem of managing proof contexts: identifying, validating, and assembling the credentials and policies that are relevant to each trust decision. This paper describes a new approach to managing proof contexts using context linking and caching. Credentials and policies are stored as certified logic sets named by secure identifiers in a shared key-value store. SAFE offers language constructs to build and modify logic sets, link sets to form unions, pass them by reference, and add them to proof contexts. SAFE fetches and validates credential sets on demand and caches them in the authorizer. We evaluate and discuss our experience using SAFE to build secure services based on case studies drawn from practice: a secure name service resolver, a secure proxy shim for a key value store, and an authorization module for a networked infrastructure-as-a-service system with a federated trust structure.

1 Introduction

Trust management deals with specifying and interpreting security policies, credentials, and relationships among entities in a system to reach an authorization decision [11]. Authorization determines whether a certain request (e.g., read, write) on one or more *objects* (e.g., file, process), is permitted by the requesting *principal*. Credentials are statements about the principals issued by concerned parties delegating the chain of trust. To enforce a given policy, each entity in the trust management system has a reference monitor (a guard) that uses credentials in conjunction with the request to infer the authorization decision.

Over time, the formal foundations of trust management systems have converged on logic-based declarative languages—*trust logic*. One prominent early example of trust management is SPKI/SDSI [21], in which participants exchange statements binding principals to names in local name spaces. Halpern *et al.* showed that the naming language in SPKI/SDSI has a logical semantics [23] and Howell *et al* provided formal semantics [25] that has roots in logic. Li *et al.* [31, 32] showed that SPKI/SDSI naming maps to a Role-based Trust (RT) language, and that the RT language in turn reduce to datalog with constraints [15], a logic language with well-understood formal properties including tractability.

However despite the flexibility and extensibility of trust logics, their application to practice is limited. We observe some key obstacles in harnessing the power of trust logics in practical distributed systems:

1. *Credential discovery.* Given a query, how to identify and assemble the tailored proof context that is relevant to an authorization decision?
2. *Credential freshness and revocation.* Scalable revocation is a major issue with the deployment of PKI based systems. The challenge is how to revoke an issued credential and propagate the changes in a timely fashion in a distributed setting?
3. *Usability.* Usability is an important goal for making trust logics practical and approachable. How to make trust logics programmable so that symbolic names are used rather than low-level identifiers for principals and objects? How to specify trust policies and perform access checks so that the system integrates naturally with the service programming environment?
4. *Federated trust.* How to name and collaboratively share resources among federated trust domains adhering to local trust policies and system constraints?

To this end we are built an integrated logical trust system called SAFE.[1] At the core of SAFE is a simple trust

[1] SAFE is an acronym for Secure Authorization for Federated Environments.

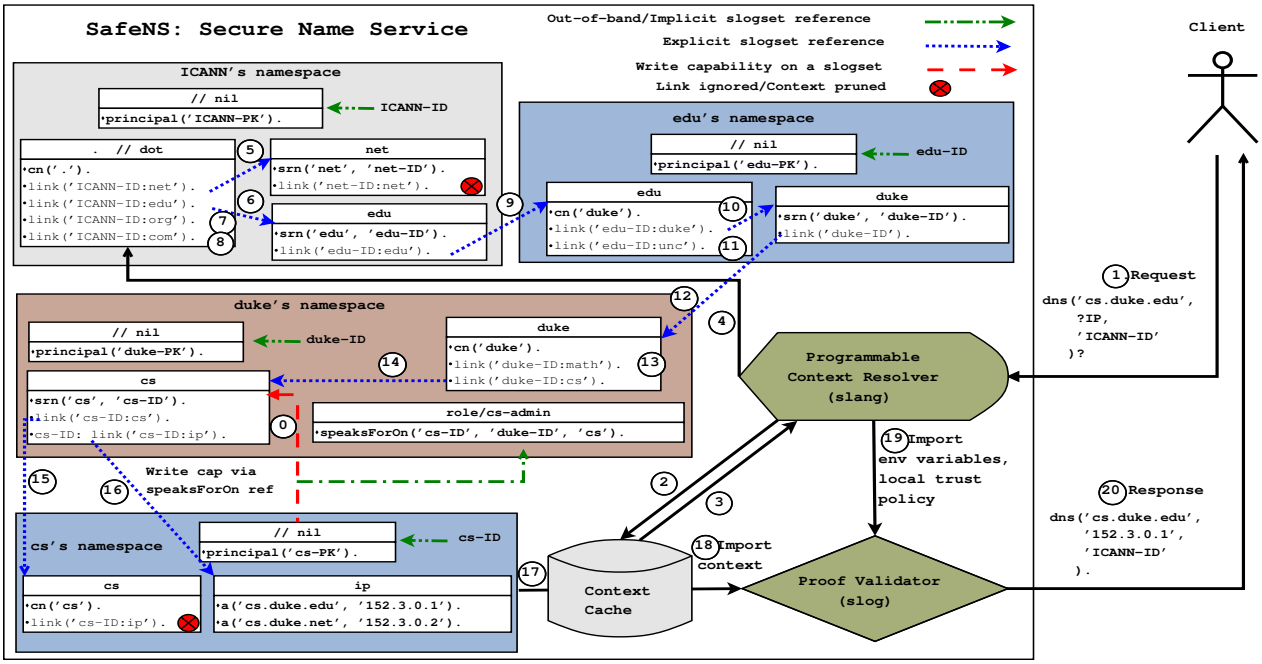


Figure 1: Workflow illustrating credential discovery, context building, context caching, and proof validation process for a secure name service—SafeNS—emulating DNSSec resolver implemented in SAFE. The credentials are issued a priori (step 0) and materialized as slogsets in a shared distributed store (SafeSets). The principal `cs` writes to a slogset owned by `duke` due to a `speaksForOn` capability issued by the owning principal (step 0). The SAFE process starts with a bearer slogset reference (`ICANN-ID`) provided by the client and traverses the credential graph via links—building a tailored context as per the resolver's programming logic written in slang (steps 1-17). The slang runtime invokes the slog interpreter importing the relevant context—and slog interpreter validates the proof based on local trust anchors and policies, and certifies the response (steps 18-20) similar to certified validation in SD3 [26].

logic (SAFE logic or slog) based on extended datalog with constraints. Slog is similar in spirit to Binder [19], SD3 [26], Soutei [37], and SENDLog [3]. These logics can capture important examples of secure network systems. For example, SD3 has been used to implement DNSSec [26] resolver, and SENDLog has been used to implement secure routing protocols that build on the declarative routing approach.

What is novel about SAFE is its approach to managing *contexts*—sets of credentials and other logic content—and the relationships among them. SAFE builds on a key concept called *set linking*, a powerful technique to organize sets of logic statements. A *link* is a meta-predicate of the credential set and serves much like a hyperlink in HTML documents. A logic set (slogset) may link to a target set by its name: the link incorporates the target as a subset, forming a union. A construction procedure integrates set linking with common primitives for delegation and endorsement, naturally materializing a graph in which each set links to the other sets needed to substantiate it. A guard specifies a context as a linked union of top-level sub-contexts (e.g., slogsets associated with the subject, object, and policy). The transitive closure of the sub-context contains all sets relevant to a given authorization decision. Set linking also facilitates flexible policy because it easy to attach policy rule sets to nodes in the graph. The cost of linking is low because common subsets are cached at the authorizer. A further advantage with linking and shared storage is that the credentials can be prefetched and cached naturally to support the materialized context sets for the future queries. Figure 1 shows a workflow of SafeNS—a secure name service emulating the DNSSec resolver implemented in SAFE(see Sec § 3.4).

In addition, SAFE offers a scripting language called "slang" that manipulates sets of authenticated logic statements (slogsets) as first class content objects with unique names. Contrast to slog, which is used as a certifying proof engine, slang is used primarily for credential discovery, certificate issuing and revocation, and tailoring proof context based on authorizer's policies. Separating the credential discovery process from proof validation is important to ensure the inference is tractable. We are also motivated to design slang to make trust logics more usable in practice—slang is declarative and makes the crypto operations transparent and provides built-in hooks to integrate with the service programming environment.

Slang provides support for higher order logical seman-

tics such as `speaksFor` and `aggregation`, which cannot be captured in slog due to lack of of function symbols and nesting. Slang offers language constructs to build and modify slogsets, publish them by name, fetch them by reference, and add them as sub-contexts to a proof context. The name space enables slogsets to be passed by reference, fetched on demand, and cached after validation at the receiver. In future exchanges the receiver may retrieve sets from its cache as needed, avoiding the need to transmit and validate them again. Slogsets are themselves stored in a shared, secure, and distributed credential store called as *SafeSets*. Each object in SafeSets is signed by its speaker to enable a third-party to verify the authenticity and integrity of the source. The shared store is also a basis for addressing perennial problems with PKI certificate management, e.g., revocation, renewal, and key rotation.

Contributions. Our research focuses on practical challenges for using logical trust in secure networked systems. Our premise is that trust logic can be as fast as PCAs and as simple identity-based access control schemes (e.g., ACLs) in the common case, while enabling rich and flexible declarative trust with a precise and rigorous logical semantics and verifiable policies. More generally, we believe that logical trust can be a fundamental enabler for a network security architecture that is richer, safer, and more flexible than the architecture in place today (e.g., X.509 and CA hierarchy). We make these contributions:

1. A high productivity programming tool for managing credentials declaratively based on language extensions to trust logics including tractable delegation with `speaksFor`, policy mobility, and server integration (Sec § 3).
2. A shared, secure, and distributed credential store that leverages the concept of set linking to organize credentials (Sec § 4).
3. An evaluation to demonstrate that SAFE is practical including the implementation of secure name service (SafeNS) and an authorization system GENI [2] (SafeGENI) in a hundred lines of SAFE scripting language (Sec § 5).

2 Logical Trust on the Network

In this section, we review the elements of trust management systems: how to name principals, objects, and logic sets; the trust requirements to satisfy in a networked system; the security assumptions and design choices that guide the implementation of SAFE. Some of the design choices are influenced by previous work on trust management—specifically, the local namespaces of SPKI/SDSI [21] and self-certifying names in [36, 35]. However one major difference of SAFE compared to SPKI/SDSI is set linking: SAFE provides an *explicit* linking capability at the level of sets of logic statements (slogsets) rather than implicit linking on names as in SPKI/SDSI. Explicit linking makes credential discovery [17] and revocation practical and scalable compared to SPKI/SDSI: the name resolution requires the authorizer to resolve the relevant certificates among a potentially large set of certificates in the *right* order; in the common case, the authorizer may simply act as a compliance proof checker putting the onus on the requester to carry the trust relationships.

2.1 Naming

SAFE relies on cryptographic keys for identifying the principals following SPKI/SDSI. A principal is a self-signed public key—i.e., a principal possesses the private key corresponding to the public key that is signed—or an attested public key by a trusted third party following the current Certifying Authorities (CAs) model. Every principal speaks indirectly through sub-principals by creating and assigning roles or by issuing `speaksFor` delegation to alleviate key rotation issues and keeping the master keypair stored securely off-line. Principals create local namespaces to keep track of resources they own, capabilities they receive, endorsements they make, or bookmark references to other principal's namespace that may contain relevant trust policies for a later retrieval.

A principal's namespace is identified globally and uniquely by a pair <`principal`, `name`> known as a *self-certifying identifier* or scid for short. Self-certifying names provide the useful property that any entity in a distributed system can verify the binding between a corresponding public key and the local name without relying on a trusted third party [35]. Self-certifying names thus provide a decentralized form of data origin authentication. Without loss of generality, the scids are defined as H_1(`principal`):H_2(`name`), where H_1 and H_2 are cryptographic hash corresponding to the tuples, giving us a fixed length scid irrespective of the key sizes and symbolic names.

In SAFE, a principal's namespace corresponds to a slogset in which credentials and policies are stated as logical statements. In addition to scid, each slogset can be identified by a secure reference (or id for short), which is formed by taking a hash of scid, i.e., $H_3(H_1$(`principal`):H_2(`name`)). An identity slogset is a special set without a name and contains principal's public key. For the identity slogset, the scid and the id are equal.

[2] GENI is a networked infrastructure-as-a-service (IaaS) with autonomous IaaS providers linked in a federated trust structure.

Objects in SAFE are identified by their scids. SAFE recognizes three types of objects: credential objects (slogsets) for which the local name is chosen by the issuer; resource objects for which the local name is autogenerated using an RFC 4122 GUID/UID; and content objects for which the local name is the hash of the contents. SAFE provides a built-in `rootId()` to extract the controlling principal name from an object name.

SAFE does not mandate a single global namespace or a central certifying authority. In SAFE, each principal is a certifying authority. Explicit endorsement and the ability to link slogsets by reference provides more flexible design choices without assuming any naming convention a priori. Where required, the hierarchical naming can be represented by aggregating `scids` explicitly. For example, a DNS request such as `cs.duke.edu` can be represented in SAFE as:
$H_1(P_.) : H_2(\texttt{edu})).H_1(P_{\texttt{edu}}) : H_2(\texttt{duke}).H_1(P_{\texttt{duke}}) : H_2(\texttt{cs})$, where $P_{\texttt{name}}$ represents the principal owning the name and H_1, H_2 are hash functions. Once $P_{.\{\texttt{dot}\}}$ is available (browsers can bootstrap trust anchors following today's practices), SAFE can be queried to infer the IP address securely subsuming DNSSec or other secure DNS implementations. Such aggregation of scids across multiple principals (with '.' as the delimiter) to form a secure compound names is known as *safe resource naming* or SRN for short. We explain the secure name resolution implemented in SAFE further in § 3.4.

2.2 Requirements

To use the trust logic in a networked system the following requirements must be met: (1) network messages can be authenticated as originating from a named principal; (2) each statement in the logic is authenticated to its named speaker; (3) each object name is securely bound to a given principal who controls the name; (4) to the extent that one party accepts or relies on another's statements, the parties must agree on the meaning of predicate symbols and names used in those statements.

Without loss of generality we meet the first two requirements by taking principal name constants as public keys (or hashes) and transporting statements in certificates signed by the named speaker, following SDSI.

The third requirement is trivially met with local namespaces, i.e., each principal hash its own object name space and requests for those objects are served only through a server controlled by that principal.

The fourth requirement is met by standards and conventions in the code. SAFE applications may define their own vocabulary of predicates. Note that common conventions are needed only for interoperability, but not for soundness. The soundness of SAFE inference requires only that statements are authentic and that the relevant name constants are unique and distinct.

2.3 Assumptions

We make the certain assumptions about the threat model, SafeSets availability, principal keypairs, and credential linking.

- The SAFE client running the inference should be a part of trusted computing base. All other components including SafeSets need not be trusted.
- SafeSets is configured to be highly available storage system. With scalable key-value stores, this requirement is easily met.
- Every principal creates sub-principals to speaks for them and stores the master key-pair securely offline.
- All delegations, speaksFor, set construction, and linking are done *explicitly* through logical assertions, i.e., SAFE does not support implicit delegations as in [2].

3 Managing Credentials Declaratively

This section presents an overview of how SAFE applications use trust logic languages (slog and slang) to build and issue credentials as slogsets, how slogsets can be linked for credential discovery, and how a set linking supports policy mobility. It illustrates with examples from SafeGENI, which is described in a related technical report [16]. The GENI trust architecture defines several classes of authority services to manage user identity and authorize user activity. These services are decentralized: each authority service may have multiple instances, and the set of instances may change over time. In addition, users may delegate various rights to one another using a capability model. SafeGENI specifies all of these structures using logic.

A notable feature of SAFE is integration of the trust logic with a scripting layer that manipulates logic content and invokes the proof engine. Slog is a tractable logic language that generates a proof locally from a supplied proof context. Who supplies the proof context and how is it assembled? The slang provides scripting constructs to build and modify slogsets from templates, link them to form unions, publish/post them (e.g., as certificates written to SafeSets), pass or fetch them by reference, and add them to query contexts for trust decisions.

An application includes slang code to construct any logic content it issues or fetch and cache logic content from other parties, assemble proof contexts, issue trust queries, and organize its credentials and policies. Each

participant in a networked system chooses the slang code that it executes: the participants exchange declarative logic content, but not scripting code.

3.1 SAFE Logic (slog)

Slog is a elementary trust logic based on constrained datalog, which is a subset of first-order logic. Logic statements in slog are written in datalog augmented with the classic **says** operator of BAN belief logic [14] and ABLP logic [27]. Statements are built up from atomic formulas (atoms) and the logical operators conjunction and implication. An atom is a predicate symbol applied to a list of parameters, which may be variables or constants representing principals, objects, or values. Every atom has a first parameter representing a principal who **says** it (the *speaker*). Consider this slog statement:

```
authorize(?Subj) :- Alice: coworker(?Subj).
```

This statement reads *"self infers authorize(?Subj), for any given subject represented by a variable ?Subj, if the principal Alice says coworker(?Subj) is true"*. The `:-` is datalog 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 whose parameters may include one or more variables (`?Subj`). A rule allows the checker 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. A *fact* is a statement with no body, and therefore no variables. The checker takes any fact in the context as true. The predicate in an atom or fact represents a property, attribute, role, relationship, right, power, or permission associated with the principals and/or objects named in its parameters.

Each atom is bound to a speaker. In the example, the atoms in the body are prefixed with a **says** operator (`:`) naming the principal `Alice`. If an atom does not name a speaker then the default speaker is `$Self`—the local authorizer who applies the statement. Note also that the speaker of an atom in the body of a rule may be a variable. Consider this rule from SafeGENI:

```
memberAuthority(?X) :- geniRoot(?Geni),
  ?Geni: memberAuthority(?X).
```

This rule reads "*self infers that ?X is a Member Authority, for any given ?X, if some principal ?Geni says it is, and self believes that ?Geni is the GENI root*". A GENI root is a principal that is accepted by members of a GENI federation to endorse authority services and IaaS providers (aggregates). The policy may include or import a fact designating a `geniRoot` trust anchor. A GENI Member Authority is a principal that is authorized to issue statements about user identity including roles, privileges, account status, and key endorsements. This rule states that the authorizer believes an assertion (`memberAuthority(?X)`) if it is spoken by any principal (`?Geni`) possessing a certain attribute (`geniRoot(?Geni)`). This form of rule is known as an *attribute-based delegation* [33].

The goals in policy rules such as these capture the meaning of delegation of trust. The delegation is restricted both by the speaker of the goal and the predicate used. For example, the rule above trusts a `geniRoot` only to endorse a `memberAuthority`, and it trusts the endorsed principal only as a `memberAuthority`. Other rules in SafeGENI delegate specific additional powers to principals with the attributes `geniRoot` or `memberAuthority`.

Function	Description
`fetch(?SetRef)`	fetch a transitive closure of slogset ref by traversing all the links
`fetchSRN(?SetRef, ?SRN)`	fetch a transitive closure of slogset ref by traversing the links as guiding by the safe resource name (SRN)
`post(?SetContents)`	post the set contents and return the slogset reference.

Table 1: Slang library functions implementing the SafeSets Client API.

3.2 SAFE Language (slang)

Slang is a simple hybrid functional-logic programming language with an extended logic syntax supporting higher order structures with nested function symbols. Slang is designed to be used as a scripting language for credential discovery, credential pruning (tailoring proof context based on authorizer's policies and the issued request), and certificate issuing and revocation. A slang program is a set of logic statements similar to the slog program but structurally akin to Prolog programs rather than Datalog programs. However, a crucial difference is that slang programs are local to the authorizer and *not transported over network* unlike slogsets. Alternatively, slang programs assist the credential discovery process and building proof context tailored to the request—but the programs themselves are not part of trust infrastructure and inference. SAFE considers the authorization decision as valid only if slog performs the inference.

A slang program permits usage of higher order constructs to process collections: lists, nested predicates, and slogsets as *first class* objects. For example, slogsets

can be manipulated as values directly by assigning them to variables and passing them as arguments to other functions.

Other important distinction from slog is that slang statements may act as functions that return values, including slogsets. In general, slang programs execute as a functional evaluation rather than inference: the evaluation follows a deterministic path with no backtracking, presuming that for each slang predicate there is at most one rule with a matching head (the common case).

The design of slang is motivated in part by our experience with building authorization system for GENI.

- Current approaches for authorization are limited: either support a high level language compromising on proof tractability (e.g., PCA [5], PolicyMaker [11], KeyNote [13], NAL [39]) or restricted language leaving the credential gathering to the applications (e.g., SecPAL [9], Soutei [37]).
- We used slog as an embedded language library from a generic purpose language and observed the impedance mismatch between language layers. For example, the slog program is passed as a string from the host language, which results in deferring the "safety" properties of slog until actual execution time.
- We observed common patterns (fetching, publishing, and renewing) for managing credentials which are handled efficiently using a high level abstraction—manipulating slogsets as values.
- Certain useful logical primitives such as `speaksFor` and `aggregation` cannot be captured at the slog layer but can be easily achieved at higher layer without compromising tractability of the logical inference [22].
- Writing slog code directly is tedious and prone to mistakes since the principals and objects are hashed values rather than simple mnemonic names. Slang makes it particularly convenient to define policies naturally through the use of lexically scoped program variables, environment variables that capture system properties, and builtin library functions that operate on slogsets directly.
- Slang also supports programming through policy templates so that policies are written once and instantiated accordingly as per the environment context and scope.
- Lastly, slang is declarative and resembles slog closely while being expressive. Slang performs traditional scripting functions: file manipulation, escaping to the host environment for program execution, and variable substitution.

```
1 defcon endorse(?IdP) :-
2   spec('endorse an identity provider'),
3   "endorse/idp/$IdP"{ // slogset name
4     identityProvider($IdP). // variable subst
5     link($Self). // link to self (geni) ID set
6   } // end of slogset definition
7 end // end of slang function
8
9 definit ?Ref := endorse('IdP-ID'), post(?Ref).
```

Code Snippet 1: Geni root endorses an IdP.

Consider the GENI example in Code Snippet 1. Geni root creates a slogset with a name ‘‘`endorse/idp/$IdP`’’ and issues simple slog statements endorsing an identity provider (IdP). The statements enclosed within { } forms a first-class slogset term extending the standard logic syntax to sets of statements. Slang supports lexical scoping and global substitution of variables. `$IdP` is a variable passed from slang to slog, which is interpolated—substituted by its value—at runtime. A slang rule tagged with a `def` keyword declares the rule as a function that returns the value of the last atom on that rule. The various slang rule types have additional behaviors to integrate with the application and with SafeSets, and to extend the scripting primitives. Slang predefines some functions with prefixes `defenv` for initializing environment variables; `defcon` for constructing slogsets; `defguard` for entry points to slang program for access checking incoming requests; and `definit` for bootstrapping the slang program. Other built-in functions that implement the SafeSets client API are shown in Table 1. Code Snippet 2 shows how a Project Investigator (PI) relies on local trust policy and bearer reference provided by the subject to determine whether the subject is a valid geni user. In this case, it is the subject's responsibility to provide a reference to a slogset, which contains a statement that the issued by IdP that the subject is a geni user. The authorizer augments the proof context constructed from bearer reference with its own local policy and invokes the slog inference. This example demonstrates the flexible and extensible authorization in SAFE in which hybrid policies are fetch on-demand or a priori at the discretion of the authorizer. See the technical report for a complete working example of GENI [16].

Slang also supports `speaksFor` and `speaksForOn` delegation in a restricted form. Our implementation of `speaksForOn` closely follows restriction delegation proposed in Snowflake project [25, 24]. However, unlike ABLP [2] and Snowflake [25] projects, we do not view `speaksFor` as a primitive form of delgation. Re-

```
defenv ?Geni :- 'geni-ID'. // hash-of-geni-PK

defcon trustStructure() :-
  spec('trust structure at the authorizer'),
  'policy/localTrust'{
     identityProvider(?X) :- geniRoot(?Geni),
      ↪  ?Geni: identityProvider(?X).
     geniUser(?U) :- identityProvider(?IdP),
      ↪  ?IdP: geniUser(?U)
   }
end

defguard isGeniUser(?Subject, ?BearerRef) :- {
     import('policy/localTrust').
     import($BearerRef). // slogset reference
     geniRoot($Geni). // substitute env var
     geniUser($Subject)? // subst slang var
  }
end
```

Code Snippet 2: Project Investigator (PI) relies on local trust policy and bearer reference passed by the `?Subject` to determine whether `?Subject` is a valid geni user.

call that slog supports attribute-based delegation with **says** as the primitive operator. In SAFE, the restricted `speaksFor` is defined as follows: if a subject `Alice` issues `speaksFor` delegation capability for `Bob`, then `Alice` grants `Bob` to write to any slogset owned by `Alice`. Similarly, `speaksForOn` restricts the capability to a particular slogset named by `Alice`. Now when `Bob` is issuing statements for `Alice`, it is `Bob's` responsibility to write to an appropriate slogset under the `Alice` namespace. The `speaksFor` and `speaksForOn` are stated as ordinary slog facts but interpreted specially by slang to achieve the desired functionality. In SAFE, we use `speaksForOn` delegation for joint ownership among principals in a group and principal/sub-principal roles, and delegating authority to a set of attributes (slogset) collectively rather than specifying each attribute individually as in ABAC [33].

Lastly, slang also provides support for `aggregation`, which cannot be supported directly in slog without loosing tractability [22]. Aggregation is useful to implement advanced features in trust logics such as threshold/manifold structures as used by SPKI/SDSI.

3.3 Set Linking and Support Sets

The power of trust logics creates new obstacles to harnessing their power in practical distributed systems. For example, authorization in decentralized federated environments involve finding the necessary credentials that satisfy the local policy for a given access control request. A key obstacle is *credential discovery*: a trust decision may require reasoning from statements drawn from various sources, requiring a method to discover and retrieve them. In general, credential discovery is the process of finding the chain of credentials that delegates the authority from the source to the requester. Credential discovery is different from the certificate path discovery in X.509 certificates [20] since credentials in trust management systems generally have more complex meanings than simply binding names to public keys. For example, a credential chain is often a DAG, rather than a linear path as in X.509.

Most previous work in trust management assumes that authorizer has already gathered all the potentially relevant credentials before a request is made and does not consider credential discovery problem further [11, 13, 9]. Even if the authorizer gathers all the credentials a priori, a crucial issue is that tailoring the credentials per query request rather than supplying all the available credentials to the proof context—since the cost of inference depends on the size of the proof context.

To make the credential discovery possible and efficient, we propose *set linking* to build trust chains by linking relevant logic sets in advance. A further advantage of our approach is that it naturally supports caching of context sets for future decisions.

The construction procedure is distributed across the participants who issue and receive credentials—slogsets containing endorsements and delegations. Each participant collects and stores the received credentials by using meta-predicate `link` to cross reference them into credential sets that it maintains. The issuer of a credential uses `link` in the set constructor to link the new set to any of its own credential sets that support its authority to issue the credential. Code Listing 1 shows `link` predicate is used as a reference to Geni root's ID set. If all issuers follow this convention then by induction the transitive closure of any given credential contains the totality of upstream credentials that an authorizer needs to validate it—the credential's *support set*. In this way, set linking naturally forms delegation chains in the credential graph. The authorizer uses its local checker to validate that these chains lead back to one or more trust anchors (e.g., `geniRoot`) according to its policies.

The bearer reference link (id) provided by the subject (`?BearerRef` in Code Snippet 2) makes the authorizer to inspect the user endorsements that she received. Linked support sets make it easy for an authorizer to obtain all credentials necessary for an authorization decision by "pulling" a credential set token passed as an argument in a request, fetching the closure of the linked subsets recursively, caching them, and adding them to the proof context. Each participant is free to organize its credentials as it sees fit, possibly across multiple sets. What

is important is that each issuer links sufficient support into each endorsement or delegation, and that each requester passes sufficient support to justify each request. The linked sets may contain a superset of what is required: the authorizer's slog engine searches the context for relevant content. We emphasize that in practice, a server finds frequently linked supporting credentials in its cache, and does not fetch or validate them again on each request.

In this way, set linking organizes credentials and policies into a DAG that facilitates discovery and assembly of proof contexts. We note that fetch is cycle-safe: it ignores any cycles, which do not affect the contents of the closure. The DAG is collaboratively editable: each node in the DAG is controlled by its owners, and changes to a set by its owners are visible in other sets that link to it. The sets are, in essence, materialized views for standard queries, in which the subset owners control what statements to include in the views.

Further, set linking naturally supports *policy mobility*: the guard policies can be defined once by the trust anchors and the authorizers can use them wherever applicable by simply fetching from SafeSets.

3.4 End-to-End Example: SafeNS

We implemented a secure name service—SafeNS—in SAFE. Using SafeNS, we emulate the DNSSec resolver in SAFE. Figure 1 illustrative the end-to-end workflow of credential discovery, set linking, context building and pruning using SafeNS as an example.

Given a name service request by the client, the browser/client-agent augments the request with a bootstrapped reference to the root (`ICAANN-ID`) slogset and passes to the SafeNS resolver. The SafeNS resolver uses the bearer reference to initiate the credential discovery process using a slang library function `fetchSRN()`. The resolver fetches the root set referenced by `ICAANN-ID` and matches the common name for the root (`cn(.)`) with the first name token `'.'` by issuing a slog query. If the slog query returns true—i.e., the safe resource name (SRN) binds/matches with the slogset local name given by the `srn()` predicate—then the search continues further following the `link` predicates until a closure is reached. The slang runtime builds a tailored context based on the SRN, and once the search completes, slang invokes slog with the relevant proof context. The slog process validates the proof based on local trust anchors (`ICAANN-ID`) and policies, and certifies the response. The workflow also illustrates the use of `speaksForOn` issued by the principal `duke` delegating ownership to the principal `cs` on a particular set named `cs`.

SafeNS resolver curtails the proof context at each stage of the credential discovery by using a constrained function `fetchSRN()` rather than `fetch()` (see Table 1). `fetch()` fetches the transitive closure name service endorsements starting from root to the proof context, which is prohibitive if not curtailed to the relevant context. It is important to note that `fetchSRN()` is implemented as a slang library function (in 30 lines of code) rather than a native implementation. The context resolver is programmable: for example, the authorizer can add custom rules to accept content only from authoritative servers located in the US region.

4 Implementation

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.

4.1 SAFE Runtime

SAFE runs as an interpreter with one or more slang programs loaded into it. The slang code can run from command-line tools or within a co-located SAFE process invoked through a REST API, or it can integrate directly with JVM applications. The code's behavior is determined not just by the slang code itself but also by the logic content passed to it. Any participant may add local rules to tailor the policies to local needs, without changing the slang program. Participants may even formulate rules and exchange them over the network as the system executes.

The interpreter is stateless, so participants may restart it and/or reload slang programs at any time: it affects only the access control for future requests. Slang programs are composable: it is easy to add code to customize the local behavior. Changing the program leaves other software and state unchanged at the site.

SAFE is implemented in Scala language including the inference engine written from scratch in about 10000 lines of code.

Slang and SafeSets offer an integrated solution for sharing authenticated logic sets in a networked system. Each authorizer's local SAFE runtime interacts with the SafeSets service to support the set abstractions of slang by fetching referenced sets on demand, caching their logic content, and assuring the freshness and validity of logic content passed to the proof engine. The SAFE runtime handles secure slogset id generation, post, fetch, and cryptographic operations automatically and transparently.

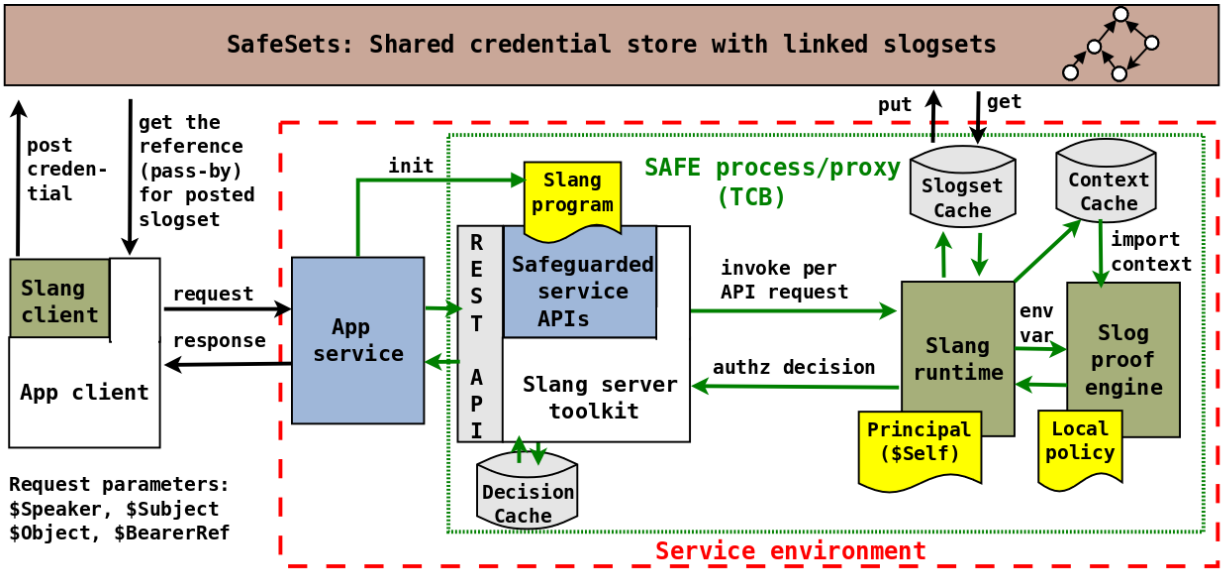


Figure 2: Server access control using SAFE. The SAFE instance runs as a separate process with a loaded program of slang that contains context building procedure, the principal's signing key (`$Self`), and the authorizer's local policies specified in slog. The server application installs slang code in the SAFE process, which registers all the `defguard` APIs for access control checks. Credentials are passed as references to signed logic sets (slogsets) in a shared distributed store (SafeSets). The SAFE process fetches slogsets on demand, validates the signature and speaker, and caches them for use by the slog interpreter.

Because the slang scripting language abstracts these details and hides them from applications, SafeSets is a replaceable component within the SAFE architecture. However, the idea of using a shared decentralized certificate store generalizes to other models for storing and authenticating the sets. In principle, logic sets could be stored in secure web directories maintained for the owning principals, or in some scenarios might be stored bare in a trusted metadata service, e.g., for use of trust logic within a single service provider domain.

When a program defines a slogset (using `defcon`), the builtin encoder consumes the meta-facts and encodes the information they contain into the selected certificate format. When SAFE fetches a certificate, the builtin decoder validates the certificate, extracts the contents, and materializes it as an in-memory slogset. The slogset represents the relevant meta-information from the certificate as logic meta-facts. These facts are available to the slog inference engine if the set is added to the context for a query.

The SafeSets service itself is implemented as a proxy shim to a scalable Riak key/value store [38]. On a post operation, the slogset id serves as the key, and the named certificate is the value. The shim checks access for post operations: it verifies that the value (a certificate containing a logic set) is signed under a public key whose hash yields the slogset id, when when hashed with the local name. The shim is implemented using SAFE itself: it is a SAFE process with slang code that invokes ordinary certificate parsing and validation, queries the meta-attributes, and performs the guard check. SafeSets clients access the Riak store only through the shim, which serves the Riak request protocol. This is a simple example of using SAFE to "safeguard" a network service transparently, as an alternative to modifying the service or integrating with a service framework.

4.2 SafeSets Certificate Store

Slang and SafeSets offer an integrated solution for sharing authenticated logic sets in a networked system. Each authorizer's local SAFE runtime interacts with the SafeSets service to support the set abstractions of slang by fetching referenced sets on demand, caching their logic content, and assuring the freshness and validity of logic content passed to the proof engine. The SAFE runtime handles secure slogset id generation, post, fetch, and cryptographic operations automatically and transparently.

Because the slang scripting language abstracts these details and hides them from applications, SafeSets is a replaceable component within the SAFE architecture. However, the idea of using a shared decentralized certificate store generalizes to other models for storing and authenticating the sets. In principle, logic sets could be stored in secure web directories maintained for the own-

ing principals, or in some scenarios might be stored bare in a trusted metadata service, e.g., for use of trust logic within a single service provider domain.

When a program defines a slogset (using `defcon`)), the builtin encoder consumes the meta-facts and encodes the information they contain into the selected certificate format. When SAFE fetches a certificate, the builtin decoder validates the certificate, extracts the contents, and materializes it as an in-memory slogset. The slogset represents the relevant meta-information from the certificate as logic meta-facts. These facts are available to the slog inference engine if the set is added to the context for a query.

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 slogset containing the required meta-attributes as facts, which are easy to specify directly in slang set constructors (`defcon`. 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. All of our experiments use the native SAFE cert format.

The SafeSets service itself is implemented as a proxy shim to a scalable Riak key/value store [38]. On a post operation, the slogset id serves as the key, and the named certificate is the value. The shim checks access for post operations: it verifies that the value (a certificate containing a logic set) is signed under a public key whose hash yields the slogset id, when when hashed with the local name. The shim is implemented using SAFE itself: it is a SAFE process with slang code that invokes ordinary certificate parsing and validation, queries the meta-attributes, and performs the guard check. SafeSets clients access the Riak store only through the shim, which serves the Riak request protocol. This is a simple example of using SAFE to "safeguard" a network service transparently, as an alternative to modifying the service or integrating with a service framework.

4.3 Server Integration

Application server frameworks can use SAFE as a proxy or invoke through REST API to check access control for client operations on the objects they server (see Figure 2). Suppose that each API method of the service has registered a corresponding guard in the slang program through `defguard`. 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. The SAFE runtime passes the request parameters for each method to the registered SAFE guards via `defguard`.

4.4 Fetching, Validation, and Caching

SAFE fetches slogsets automatically on first reference to a token in the slang program. The client side code performs a cycle-safe recursive fetch and the requests are parallelized using a thread pool for reduced latency. After each subset is fetched, SAFE validates the signature, parses the certificate, and authenticates the stated speaker of each 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 as *slogset cache*. The fetch checks the set cache for each subset token encountered during the recursive fetch.

When a sub-context is assembled and dispatched to the inference engine, SAFE *renders* it to a internal context format and caches it in a *context cache*. A rendered context set is flattened, then indexed to speed up the inference engine, which must search for rules with heads matching each goal. The expiry date on the context cache is set to the *lowest* expiry dates from the collections of sub-contexts that are assembled. SAFE tracks the period of validity internally to expunge any expired content from the caches. A leaf subset expires from the slogset cache at the expiration time of its containing certificate. We also added support to invalidate a cached context at the earliest expiration time of any statement it contains, so the proof engine sees only fresh logic content. If an expired certificate is reissued, then a fetch pulls the fresh certificate from the store automatically.

We enhanced server integration by adding initial support for a query *decision cache* on the Web server side. The result cache optimizes repeated operations by a given subject on a given object, by avoiding the inference check entirely for a configurable time. We do not report results from decision cache.

4.5 Certificate Management

SAFE and SafeSets address a number of longstanding challenges for certificate management.

Renewal. An issuer may renew an expired certificate by posting the renewed certificate to SafeSets with the same identifier, overwriting the expired certificate. If a SAFE authorizer encounters an expired set it uses the

function	compute hash	verify signature	sign a set	parse a set	null inference	fetch a set	post a set
latency (ms)	0.13	0.56	12.14	2.2	0.09	7.8	27.4

Table 2: Micro-benchmarks of basic operations in SAFE on 1kB of payload per certificate. Fetch and post costs are network latencies over WAN for reading and writing/updating a slogset to SafeSets. Keys are 2048-bit RSA keys. Hash function is set to SHA-256. The *null* inference is the minimum latency penalty for querying slog through slang.

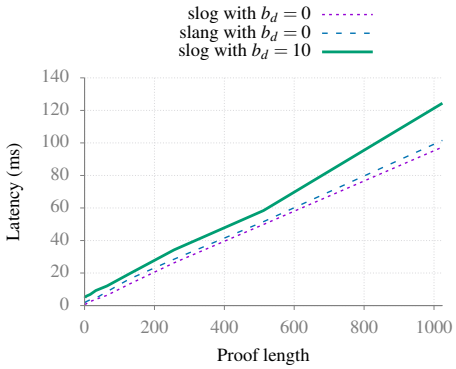


Figure 3: Cost of inference with varying proof length and degree of backtracking b_d. The latency measurements show that the inference cost scales linearly if b_d is kept low. The plot also shows the overhead of calling slog from slang program is minimal ($< 5\%$).

identifier 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 issuer may control the expiration times to bound the time that a set remains in an authorizer's cache by setting the `refresh` time on the published set. An authorizer may refresh sets in its cache at its discretion, even if they have not expired.

Rotation. SAFE with SafeSets names principals by their public key hashes. If a principal loses or rotates its key-pair, then sets that incorporate the stale key in their set identifiers must be regenerated. To avoid potentially expensive set identifiers regeneration, SAFE advocates each principal to create sub-principals which `speaksFor` the master on a given role (e.g., signing, encryption).

The SAFE approach to managing credentials also raises some potential concerns which we discuss in § 6.

5 Evaluation

We evaluate SAFE on mix of cache configurations, micro-benchmarks, and real applications. We seek to answer three questions:

Q1. Does SAFE achieve acceptable performance? How SAFE compares to ACLs and capability based access control where policies are attached directly to entities rather than managed independently through SAFE reference monitor? What is the overhead of invoking slog through slang?

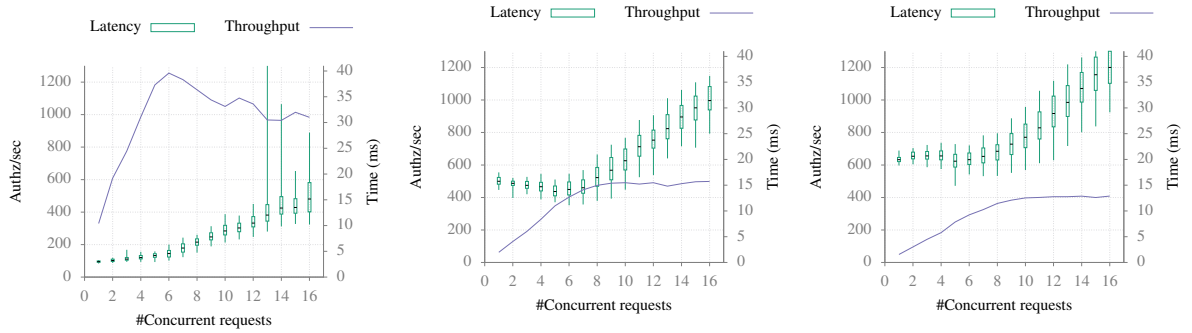
Q2. Are trust logics practical for use and deploy in real applications? What is the programming effort required to build secure applications using SAFE as the foundation for trust management and access control?

Q3. How does set linking and context caching improve performance across space (cross-sharing of common slogsets among multiple simultaneous queries) and time (caching frequently accessed slogsets)?

All our authorizer experiments are conducted on eight core Intel Xeon CPU E5520 @ 2.27GHz processor with hyper-threading enabled and 8 GB of RAM running CentOS 5.10. The SafeSets cluster consists of four VMs, each with a single core Intel Xeon CPU E5620 @ 2.40GHz processor and 1GB of RAM interconnected by a 1Gb network, all running Ubuntu 14.04. The authorizer access the SafeSets store over WAN. We use unmodified Riak 2.0 [38] as our key-value store for SafeSets guarded by SAFE as a proxy "shim" to authorize writes to a slogset.

5.1 Micro benchmarks

To answer the first question, we use micro-benchmarks to evaluate SAFE performance. Table 2 show the overhead of basic operations in SAFE on a 1kB of payload per certificate. The slogset identifiers use *SHA-256* for hashing and base64 for encoding, which result in fixed 44 byte strings. For signing, we use 2048-bit RSA keys. The *null* inference is the minimum latency penalty of querying slog inference through slang. The overhead is proportional to the size of proof context size and the number of environment variables which are globally substituted from slang to slog. A fetch here is a single slogset retrieved from SafeSets without traversing any links. A fetch is verified for its authenticity—by verifying the signature on the slogset—at the authorizer. On the contrary, posting a slogset requires the client to sign its contents

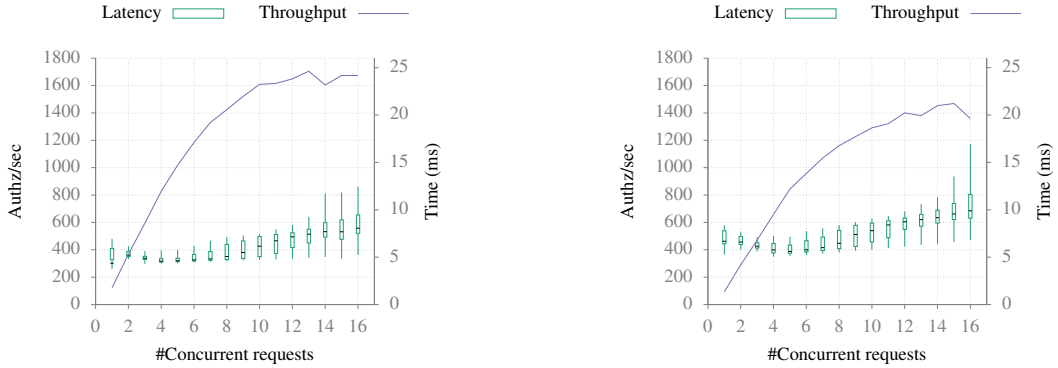


(a) SafeSets write without authorization

(b) SafeSets write without authorization but parsed on the server (baseline)

(c) SafeSets write with SAFE authorization enabled

Figure 4: Performance comparison of issuing a *write* to SafeSets store (a write is a post with slogset signing excluded).



(a) SafeNS proof validation cost for delegations of length 4

(b) SafeGENI authorization cost with delegations of length 6

Figure 5: Performance comparison of SAFE applications. Subplots 5a and 5b show the performance of SafeNS and SafeGENI systems respectively.

and send it to the SafeSets server over the secure channel to prevent replay attacks. The SafeSets server nodes are guarded by SAFE to determine whether the requesting principal has write access to the slogset either directly or through `speaksFor*` capability. For the micro-benchmark, both fetch and post including the cost of verification and signing the slogset. Table 2 shows latency of post is 4X times the latency of fetch: post is expensive since each post is idempotent and performs read-modify-update to a slogset.

To analyze the cost of inference, we simulated delegation chains which the number of unifications exactly matches the proof length. We measure the latency by varying the length of number of unifications matching the goal from 1 to 1024 and the controlling the degree of backtracking, b_d. When b_d is set to zero, we have no backtracking and the length of the proof chain is linear in terms of the unified goals in the input query. Setting b_d to zero approximates proof-carrying-authorization (PCA [8]) where the length of the input is exactly the length of the proof chain. In our applications of SAFE, we observed that backtracking scenarios may occur with attribute-based-delegation, where the principal on a goal is variable and the proof context have multiple rule heads matching on the same goal. However, SAFE uses indexing on multiple parameters which often reduces the b_d to a small value. Index optimization is a work in progress. In our next experiment, we set b_d to 10, i.e., each goal can have at most 10 possible rule heads matching with the goal and the length of the proof chain may grow exponentially with the input size. Figure 3 shows the latency measurements of when varying proof chain length and b_d. When b_d is zero, latency grows linearly with the proof length, which is expected. However, when b_d is 10, it is interesting to note the latency remains linear and only deviates from a linear scale at when the number of unifications matching the goal (proof length in this case) is 600 or above. The result shows that set linking and tailoring proof contexts is important to keep b_d low, which will in turn help to scale the inference cost linearly with the size of the input.

The latency costs in Figure 3 show that SAFE infer-

	# Rules	# SLOC
SafeSets	2	15
SafeNS	7	40
SafeGENI	30	110

Table 3: Analysis of programming effort for building declarative trust applications in SAFE.

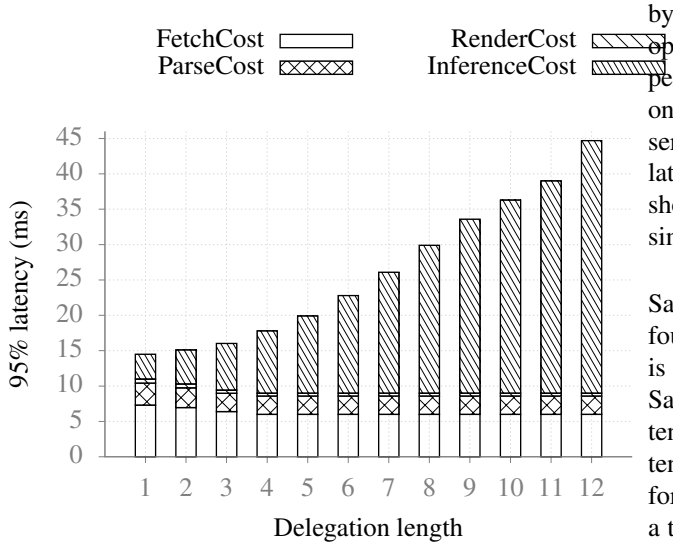


Figure 6: End-to-end authorization costs of SafeGENI with varying delegation lengths.

ence takes 0.1 ms per unification, which is competitive with respect to identity based ACLs, which needs only one fact checked. Further, the plot shows that comparison of inference costs when input is feed directly to a slog interpreter vs. executed from the slang program. Recall that slang program will in turn invoke slog interpreter after substituting all the environment variables. The plot shows that overhead of calling slog from slang is minimal ($< 5\%$).

5.2 Applications of SAFE

We built authorization systems for three practical applications in SAFE. Table 3 shows the modest effort required to build applications using SAFE.

First, SafeSets uses SAFE as a proxy "shim" guarding write access to slogsets. The post authorization for SafeSets involves validating the speaker that signed the set on the server to determine the set ownership. SafeSets is also a good example to illustrate the application of `speaksFor` and `speaksForOn` predicates, which are implemented at slang as discussed in § 3.2.

Figure 4a show the performance comparison of SafeSets write (a write is a post with slogset signing exclued on the client) without any authorization performed on the server, where as Figure 4b show the performance comparison without any authorization but slogset parsed by the server (baseline). Figure 4c show the performance comparison of SafeSets write with server authorization using SAFE.

The measurements show the peak throughput drops by 63% due to parsing overhead—our parser is not optimized—and the median latency increases by 50% per write operation. If we compare the parsed slogset on the server with the cost of authorization, then we observe that peak throughput drops by 8% and the median latency increases by 9% per write operation. These plots show that slog validation overhead is less than 10% for simple proofs that emulate ACLs.

Second, we implemented secure name service—SafeNS, discussed as in § 3.4. For delegations of length four (the average request sub-domains for a DNS query is three), the 95% latency of proof validation cost of SafeNS is 6ms, which is a fraction of DNS lookup latency (in the order of tens of ms). Figure 5a shows the latency and throughput measurements of a proof validation for a NS query with delegation length four. We achieved a throughput of 1600 auth ops/sec on our test suite with one authorizer node.

Third, we prototyped authorization system for GENI, which is a networked infrastructure-as-a-service (IaaS) system with autonomous IaaS providers linked in a federated trust structure. GENI serves as a full-featured network trust example that includes distributed objects (groups and slices[3]) with hybrid capability-based access control, multiple object authorities, authority services for group membership and federated identity management, and a common root trust anchor that endorses the authorities and member sites.

The GENI trust architecture defines several classes of authority services to manage user identity and authorize user activity. These services are decentralized: each authority service may have multiple instances, and the set of instances may change over time. In addition, users may delegate various rights to one another using a capability model. SafeGENI specifies all of these structures using logic and implemented in 110 SLOCs of slang. For delegations of length six, the 95% latency of authorization cost for SafeGENI is under 10ms with a throughput of 1400 authz ops/sec. The end-to-end authorization cost including fetching from SafeSets, validating the sets and ripping the crypto, building a context cache is under 20ms (see Fig 6). Most GENI delegations are smaller than length 6. Figure 5b shows the latency and throughput measurements of authorization costs of SafeGENI.

[3] A slice is a set of resources requested by an user

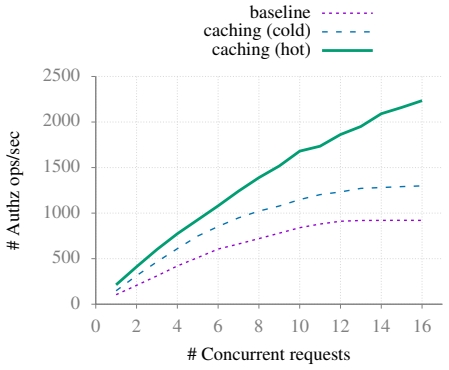


Figure 7: Throughput measurements for SafeGENI benchmark with N set as 1024 and M set as 4096. The measurements show caching proof contexts can help to achieve linear scaling of throughput.

5.3 Impact of caching

To measure the impact of caching and context linking on performance, we benchmark SafeGENI using a standard GENI workload with a mix of N users and M resources. We set up delegation chains so that any user is at most $log_2(N)$ delegations away from accessing a resource. We measure the throughput and latency of the mix under three scenarios: (i) Baseline case: all certificates are processed in their entirety for each request with no caching involved. This includes retrieving all certificates from in-memory, validate crypto and speakers, render them to set cache, merge them to context cache, and querying the inference. (ii) Cold caching with monolithic contexts: the raw certificates are cached in memory but the slogset cache and the context cache are build on-demand. These context caches and slogset caches are monolithic in that their life span tailored to a given request. (iii) Hot caching with set references and proof context cache enabled. Here we cache the proof contexts which enables shared credentials among queries is readily available through the context cache.

Figure 7 shows the peak throughput measurements for the three scenarios with N set as 1024 and M set as 4096. The throughput for baseline case flattens out fast as expected since each request is processed in its entirety, i.e., by validating the certificates, ripping the contents, and evaluating the query. Caching the certificates improves throughput by 40%. However, the throughput flattens out after 10 concurrent requests. With hot caching, the proof contexts are shared across the queries resulting shared trust policies and credentials avoiding the re-rendering to proof context. The measurements show that throughput scales linearly with hot contexts and demonstrates the useful of caching proof contexts.

6 Discussion

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 slogsets 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 [42]) for certificates they post to SafeSets representing the result of access decisions. Accountability is an active research topic.

Confidentiality. Synthesized identifiers raise the issue of confidentiality of policy rules and other logic material stored in SafeSets. If an entity wants to protect a confidential logic set it may salt the label: it is infeasible to guess a hashed identifier 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 slogsets 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.

7 Related Work

As flexible and extensible trust logic for federated network systems, SAFE build upon a wealth of prior results. Thus, for SAFE, the closely related work covers the entire fields of study—including authorization logic [9, 41, 19, 26, 8, 30], trust management [13, 12], proof-carrying authorization [5, 28], and scalable storage systems [18].

In general, trust logics apply common axioms of ABLP access control calculus [2]. In particular, every entity *controls* its own beliefs through axioms are known as *Hand-off* and *Bind* respectively [1]. They enable sound trust policies and mobility of policy rules.

Many of these logics offer features that are not present in slog: our research goal is not to advance trust logics, but to facilitate their practical use. While at present we see little need for features such as threshold/manifold structures or negation, slog could grow to incorporate them without compromising tractability, following SecPAL [9] and NAL [39]. However, SAFE supports a simple negation restricting only queries to contain `not` predicate to allow `deny` conditions (blacklists).

Some logics use (e.g., `speaksFor` and `speaksForOn`) as primitive axioms to delegate trust [2, 24, 39]. On the contrary, SAFE represents delegation with rules in datalog-with-`says`, and uses `speaksFor` predicates only for joint ownership or to authorize third-party attributions, e.g., for service proxies or portals that are trusted to issue statements for which the named speaker is another party.

The paper uses set linking and linked contexts to address the challenge of assembling the context for a proof. One option is to require the caller to submit the proof to the authorizer, as in Proof-Carrying Authorization [8]. PCA merely shifts the burden of assembling the context to the caller. Our premise is that PCA is unnecessary for a simple trust logic like datalog-with-`says` and careful context management: constructing a datalog proof from a small proof context can be fast. In contrast, the AF logic that underlies PCA is intractable in the general case. The other option is bearer credentials that are only verified by the specific *target* service rather than any authorizer as proposed by Macaroons [10] using HMACs.

More fundamentally, whether or not PCA is used, the caller must identify the relevant credentials to send with a request, or else the authorizer must obtain them by some other means. Previous approaches to credential discovery are based on a distributed query model (e.g., [34, 6, 7]). In SPKI/SDSI, the name resolution [17] requires the authorizer to resolve the relevant certificates among a potentially large set of certificates in the *right* order. On the contrary, explicit set linking makes credential discovery scalable and practical. Many previous works ignore the problem and presume that the caller will identify the correct credentials and pass them in the request. In these systems, the receiver/authorizer validates and checks the request credentials even if it has already received them for a previous request. In contrast, linked context sets naturally support caching and pass-by-reference for credentials.

SafeSets facilitates certificate sharing via a highly available shared store. The early X.500 model proposed a distributed certificate repository, as discussed by the SPKI/SDSI authors [21], who judge it to be impractical. A key difference is that SafeSets links certificates by set identifiers, and does not rely on global principal names other than public keys, as advocated by SPKI/SDSI. Various other systems have proposed certificate storage for use in credentials-based authorization (ConChord [4], CERTDIST [40]) using various indexing and naming schemes, but they do not support set linking. SafeSets supports secure unforgeable set identifiers for a key/value store by qualifying them with a public key hash. A similar technique has been used in many DHT applications [36, 29].

8 Conclusion

SAFE is a trust management system that uses a declarative trust logic to represent policies, endorsements, and delegations. What is novel about SAFE is the integration of the trust logic with a scripting language ("slang") and shared storage abstraction for authenticated logic content. These elements work together to simplify and automate many aspects of networked trust. In the implementation for this paper we materialize logic sets as signed certificates, as in SPKI/SDSI, and store them in a scalable certificate store called SafeSets. Each stored certificate is named by an identifier suitable for indexing and caching linked certificates or logic sets.

We use this combination to address three fundamental problems: how to identify the content that is relevant to a given trust decision, how to manage the flow of credentials through the system including caching, and how to incorporate updates to outstanding certificates. Experience with SafeNS and SafeGENI shows that the approach is practical for a complex network trust system.

9 Acknowledgemnts

The first author is partially supported by NSF grants OCI-1032873 and CNS-1330659 for this work.

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] S. Ajmani, D. E. Clarke, C.-H. Moh, and S. Richman. Conchord: Cooperative sdsi certificate storage and name resolution. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 141–154, London, UK, UK, 2002. Springer-Verlag.

[5] A. W. Appel and E. W. Felten. Proof-carrying authorization. In *6th ACM Conference on Computer and Communications Security*, pages 52–62, 1999.

[6] L. Bauer, S. Garriss, and M. K. Reiter. Distributed proving in access-control systems. In *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, SP '05, pages 81–95, Washington, DC, USA, 2005. IEEE Computer Society.

[7] L. Bauer, S. Garriss, and M. K. Reiter. Efficient proving for practical distributed access-control systems. In *Computer Security – ESORICS 2007: 12th European Symposium on Research in Computer Security*, volume 4734 of *Lecture Notes in Computer Science*, pages 19–37, Sept. 2007.

[8] L. Bauer, M. A. Schneider, and E. W. Felten. A general and flexible access-control system for the web. In *Proceedings of the 11th USENIX Security Symposium*, pages 93–108, August 2002.

[9] 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.

[10] A. Birgisson, J. G. Politz, Úlfar Erlingsson, A. Taly, M. Vrable, and M. Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Network and Distributed System Security Symposium*, 2014.

[11] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *1996 IEEE Symposium on Security and Privacy*, pages 164–173, May 1996.

[12] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on*, pages 164 –173, may 1996.

[13] M. Blaze, J. Ioannidis, and A. D. Keromytis. Experience with the KeyNote trust management system: Applications and future directions. In *Trust Management, First International Conference, iTrust 2003*, volume 2692 of *Lecture Notes in Computer Science*, pages 284–300, 2003.

[14] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Trans. Comput. Syst.*, 8(1):18–36, Feb. 1990.

[15] S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog (And Never Dared to Ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

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

[17] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI, 2001.

[18] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007.

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

[20] Y. Elley, A. H. Anderson, S. Hanna, S. Mullan, R. J. Perlman, and S. Proctor. Building certifications paths: Forward vs. reverse. In *NDSS*. The Internet Society, 2001.

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

[22] M. Grohe. From polynomial time queries to graph structure theory. In *Proceedings of the 13th International Conference on Database Theory*, ICDT '10, pages 2–2, New York, NY, USA, 2010. ACM.

[23] J. Y. Halpern and R. Van der Meyden. A logic for sdsi's linked local name spaces. *Journal of Computer Security*, 9(1):105–142, 2001.

[24] J. Howell and D. Kotz. End-to-end authorization. In *Proceedings of the 4th Conference on Symposium on Operating System Design & Implementation - Volume 4*, OSDI'00, pages 11–11, Berkeley, CA, USA, 2000. USENIX Association.

[25] J. Howell and D. Kotz. A formal semantics for spki. Technical report, Hanover, NH, USA, 2000.

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

[27] 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.

[28] C. Lesniewski-Laas, B. Ford, J. Strauss, R. Morris, and M. F. Kaashoek. Alpaca: Extensible authorization for distributed services. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 432–444, New York, NY, USA, 2007. ACM.

[29] J. Li, M. Krohn, D. Mazières, and D. Shasha. Secure untrusted data repository (sundr). In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 9–9, Berkeley, CA, USA, 2004. USENIX Association.

[30] 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, Feb. 2003.

[31] 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.

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

[33] 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.

[34] N. Li, W. H. Winsborough, and J. C. Mitchell. Distributed credential chain discovery in trust management: extended abstract. In *8th ACM conference on Computer and Communications Security*, pages 156–165, 2001.

[35] D. Mazières and M. F. Kaashoek. Escaping the evils of centralized control with self-certifying pathnames. In *Proceedings of the 8th ACM SIGOPS European Workshop on Support for Composing Distributed Applications*, EW 8, pages 118–125, New York, NY, USA, 1998. ACM.

[36] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, SOSP '99, pages 124–139, New York, NY, USA, 1999. ACM.

[37] A. Pimlott and O. Kiselyov. Soutei, a logic-based trust-management system. In *Proceedings of the 8th International Conference on Functional and Logic Programming*, FLOPS'06, pages 130–145, Berlin, Heidelberg, 2006. Springer-Verlag.

[38] Riak key value store. `http://docs.basho.com/riak/`, 2015.

[39] F. B. Schneider, K. Walsh, and E. G. Sirer. Nmazieres:1999:skm:319151.319160,exus authorization logic (NAL): Design rationale and applications. *ACM Transactions on Information System Security (TISSEC)*, 14, May 2011.

[40] S. Sevinc, L. Peterson, T. Jim, and M. Fernández. An emulation of geni access control. In *Proceedings of the 2Nd Conference on Cyber Security Experimentation and Test*, CSET'09, pages 7–7, Berkeley, CA, USA, 2009. USENIX Association.

[41] 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.

[42] 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.