A Guided Tour of SAFE GENI [1]

Jeff Chase and Vamsidhar Thummala

Technical Report CS-2014-002
Department of Computer Science
Duke University
June 2014
Revised 9/3/14

Abstract

GENI is a federated (multi-domain) Infrastructure-as-a-Service (IaaS) system. SAFE GENI is an implementation of the GENI architecture using SAFE, an integrated system for logical trust. This paper gives an overview of the SAFE GENI implementation. It is useful both as an introduction to the GENI architecture and as an example to illustrate how to use SAFE to program network trust.

1 Introduction

Over the last few years GENI has evolved from a collection of research prototypes into a production deployment [4]. GENI is well-suited as a driving example for network trust management. The GENI architecture is best understood as a federation of autonomous IaaS providers ("aggregates") and related services linked by various trust relationships. Each provider has various policies governing access by its clients. Some of the policies are local and some are shared by all members of the federation. The access policies involve various endorsements and delegations of trust among the members of a GENI federation. In this respect GENI is representative of other federated services.

SAFE is a trust management system that uses a declarative trust logic to represent policies, endorsements, and delegations. Logic is useful as a specification tool for the GENI architecture, independent of the implementation. With SAFE or other suitable programming tools, logical trust also enables a practical and concise implementation of key elements of the architecture using declarative policy. Declarative implementation offers important benefits: it enables GENI federation deployments to use a wide range of trust structures and policies specified in declarative logic, using the same software base. The policies and trust structure may evolve over time without modifying the software.

An earlier paper [5] makes the case for a GENI implementation based on trust logic, and includes a more complete treatment of the concepts and rationale of the GENI trust architecture as a solution for federated infrastructure services and community clouds. This paper extends that work with a more complete implementation based on the new SAFE system. The design of SAFE was driven in part by the lessons of the GENI experience and challenges encountered in our earlier research.

SAFE GENI implements the GENI architecture, but it is not compliant with existing deployed software for GENI. The deployed GENI software uses a structured URN naming standard to encode various attributes and relationships into names, together with a few custom certificate formats, and custom code to process them. (See Section 8.2 and 8.1.) Our system uses the SAFE certificate format to transport general logic statements, and it does not use the naming system for trust. We call it SAFE GENI to distinguish it.

This tour of SAFE GENI focuses primarily on the structures to manage user identity and authorize user activity in GENI. Since these functions are system-wide in a federated (multi-domain) IaaS system like GENI

[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.

it is desirable to factor them out of the infrastructure providers (aggregates) and into common services that mediate interactions among users and aggregates. For this purpose GENI defines several classes of "authority" services that coordinate identity management and authorization. These services are decentralized: each authority service may have multiple instances, and the set of instances may change over time.

The SAFE GENI example shows how to build the GENI authorities using SAFE. Since the authority services do little more than produce and consume credentials, we can implement them entirely with SAFE and a Web server framework. We also show how to implement access control policies for GENI aggregates, and discuss the role of SAFE for logic-based resource representations in GENI aggregates.

2 Using SAFE for GENI

The goal of the SAFE project is to promote trust logic as a practical high-productivity programming tool for secure networked systems like GENI. We believe that adequate support for logical trust can dramatically accelerate their development and evolution. More generally, we believe that logical trust is a fundamental enabler for a network security architecture that is richer, safer, and more flexible than the architecture in place today.

SAFE provides a trust logic (SAFE logic or "slog") to represent credentials and policies, a logic-based scripting language ("slang"), a slang interpreter and slog inference engine, and a decentralized store (SafeSets) to share credentials and protect them cryptographically. This paper presumes some familiarity with the basics of trust logic and the SAFE system. A companion paper [6] introduces these topics and some terminology used in this paper. One goal of this paper is to provide a detailed example of how to use SAFE to build a real system: SAFE GENI.

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*. Logic sets represent credentials, delegations, endorsements, and policies. Slang programs may construct 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.

The core of SAFE GENI is a collection of slang programs that operate on logic sets. Slang programs create logic sets using *set constructors*. Each slang set constructor specifies the statements in the set together with a *label*, which is a user-defined string used to derive the set token. The set constructor specifies the label as a template: it may include environment variables whose values are substituted at runtime to generate the label value. SAFE qualifies the label value with the issuer's public key to form a secure global name, and hashes it to a fixed-width token. Currently a token is a 256-bit SHA hash encoded as a 52-byte base64 string.

Slang programs use the set tokens to link logic sets and pass them by reference. The sets are stored in SafeSets as linked collections of digitally signed certificates. SAFE fetches and validates a certificate when the slang program references a containing logic set by its token. After validation SAFE extracts the semantic content of a certificate into an in-memory logic set. Sets created in slang or imported from SafeSets are cached in an in-memory *set cache*. The slang programs deal only with the semantic content of certificates: the SAFE runtime encodes and decodes logic material, handles cryptographic operations, and performs fetch, retrieval, and caching automatically and transparently.

SAFE runs as an interpreter process with one or more slang programs loaded into it. Slang is a string-oriented functional scripting language that extends the logic syntax of slog. Users may launch slang programs from the command line, e.g., to run a named program in batch mode. Command line arguments are passed into the slang program as named environment variables.

Servers run SAFE as a companion process and use a REST API to invoke it through a protected socket. To use SAFE for access control the server initiates a logic query (a *guard*) to check access permission for each incoming request. 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 framework

may invoke the guard before invoking the application method for a request. Ideally there is no change to the application itself. Some applications—e.g., the GENI authority services—also issue credentials by invoking a separate *post* operation in the slang program after the method completes.

The SAFE GENI system provides many examples of these concepts in action.

3 Principals and Their Keys

Every GENI principal has a keypair with a registered public key. Users use their keypairs to authenticate their requests to servers. Users may also run utility programs that transfer privileges to other users; one example is granting another user a capability to access a slice. Similarly, an authority server may issue credentials to users under its keypair.

In SAFE GENI all credentials are represented as logic sets posted to SafeSets. Slang programs issue credentials on behalf of the controlling principal by constructing a logic set and posting it. To issue credentials the SAFE process runs with access to the controlling principal's signing key and files.

3.1 Generating a Keypair

Each user or service administrator may generate its keypair by running this simple slang program from a command line.

```
definit mintKeyPair() :-
   spec("Mint a new RSA keypair using a key size of 4096 bits"),
   SelfKeyPair := generateKeyPair("RSA", 4096),
   writeKeyPair(@("/home/user/keypair.pem"), SelfKeyPair).
```

The entire program is a single `definit` rule. A `definit` rule evaluates automatically when the program is loaded. Every subgoal must succeed, or else the program fails. Most of the work in this simple program is done by the builtin function predicates `generateKeyPair`, `writeKeyPair`, and `@`. SAFE provides a library of builtin predicates for standard crypto and networking functions available in Scala, Java, and other JVM languages. The `spec` predicate is a builtin for embedding a string description (a comment) in a rule: `spec` does nothing and always returns true.

The second goal of this `definit` rule introduces the shorthand syntax for a variable assignment atom in slang and slog. It binds a logic variable and evaluates to its value, which is true if the value does not begin with the string "false".

This example illustrates that slang uses a form of "duck typing" of string values. All values are strings. Slang includes various builtin functions to manipulate strings in particular formats and interpret them in a particular way. Any string in the expected format is legal as a parameter. For example, there are specific predefined formats for strings representing names and keying material in the SAFE system.

The format and content of each string may convey dynamic type information. For example, the `generateKeyPair` builtin returns a string in a particular format indicating that it is a list with two string members: "[publicKey, privateKey]". The elements in the list are strings representing the two halves of the keypair in base64 encoding. The `writeKeyPair` builtin succeeds only if the input has the expected format for keying material to be encoded into a pem file.

Similarly, the builtin `@` operator takes a document location, which is typed by its format and extension. In this case, it is a local file pathname of a pem file. SAFE can also fetch documents from secure URLs: for example, slang programs can bootstrap trust from the existing Internet by fetching public keys of their peers

from websites. SAFE has builtin support for various extension encodings, including keying material (.pem), slang program files (.slang), and signed logic sets (.slog).

Note: one could instead use `ssh-keygen` or some other preferred tool to generate the keypair. What is important is that the keypair is available in some recognized format, such as a pem file.

3.2 Generating an Identity Certificate

Once the keypair is generated and saved locally, it is a good idea to register it by posting an identity certificate to SafeSets. The identity certificate may be a standard X.509 identity certificate, and may be (but need not be) self-signed. Any valid certificate may be used. SAFE can generate an X.509 identity certificate using the following slang code:

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

defcon identitySet() :-
  PublicKey := publicKey($SelfID),
  {
     encoding("X.509").
     subject(PublicKey).
     subjectAltName("email", $Email).
     subjectAltName("uri", $Uri).
     notBefore(XXX);
     notAfter(YYY);
     label("");
  }.
```

This program illustrates the use of a `defenv` rule to initialize a standard environment variable `$SelfID` to the local principal's keypair retrieved from a local file. SAFE uses `$SelfID` to sign exported logic sets and generate secure tokens. A slang program must initialize `$SelfID` to a registered keypair before posting any content.

The `identitySet` rule is a simple set constructor. It is an ordinary function, but the `defcon` rule tag indicates that its return value is a string representing a set token. The {} goal declares the slog statements in the constructed logic set.

The statements in a set may include reserved meta-facts that drive its certificate encoding. If the set is exported, the `encoding` meta-fact selects the X.509 encoder: the set is exported as an X.509 certificate. In this identity set example, all of the statements in the logic set are predefined meta-predicates understood by the builtin X.509 encoder. The `subject` parameter contains the full public key of the subject of the certificate. The `subjectAltName` parameters are required by the X.509 specification. The period of validity has a default value but may be controlled using the `notBefore` and `notAfter` predicates.

This slang program fills in the `subjectAltName` values from the environment variables `$Email` and `$Uri`. These may be defined on the command line: any "name=value" pairs on the command line are passed into the slang code as environment variables.

When a program posts a logic set, 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 logic set. The logic set represents the relevant meta-information from the certificate as logic meta-facts using the builtin predicates shown above, and others like them. These facts are available to the slog inference engine if the set is added

to the context for a query. SAFE also tracks the period of validity internally to expunge any expired content from the caches.

The last meta-fact to consider in the identity set example is `label`, a builtin that associates each locally constructed set with a string label. The label must be unique among all locally constructed sets: sets with the same label are silently merged. In this case the label is an empty string—an important convention for identity sets encoded as identity certificates. (See Section 3.3.)

3.3 Posting and Secure Tokens

In SAFE GENI each principal's identity certificate (identity set) is posted to SafeSets and indexed by its key hash—the hash of the subject's public key. With this convention participants may reference one another's public keys using hashes, which are much more compact than the keys themselves. Anyone who knows a principal's key hash may retrieve its identity certificate and extract the full public key. This property is useful because the full public key is needed to authenticate statements made by that principal.

Here is a fragment of slang code to post the identity certificate, when combined with the `identitySet` constructor code in Section 3.2.

```
definit main() :- postIdentitySet().

defpost postIdentitySet() :-
  [identitySet()].
```

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` example above. In this example the list has only a single element: a reference to the identity 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. To derive the token it concatenates the public key hash of `$SelfID` with the label, and computes a SHA hash of the result. In this case the set's label is an empty string, so the second hash is not computed: the token is just the public key hash of `$SelfID`. In this way the token for a principal's identity certificate is the hash of its public key.

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.

4 Identity

So far each principal has a keypair, an identity certificate, and a name, which is its key hash—the hash of its public key. These are sufficient for principals to authenticate their communications with one another, including authentication of logic sets by signing.

But how to know whether to trust a principal? So far we have only the identity certificate, which may be self-signed. A self-signed certificate conveys no useful trust information at all. Even an identity certificate signed by a Certifying Authority asserts only a global "distinguished name" for the principal.

The creators of SPKI/SDSI argued in RFC 2693 [7] that names are neither necessary nor sufficient as a practical basis for identity and trust in multi-domain systems. They further argued that the identity of a

principal is equivalent to the sum of the statements made by other principals about its public key. In SAFE any principal may issue any slog logic statement naming the public key of another as a parameter. Such a statement asserts a fact or attribute of the principal named by the key. We may refer to it as an *endorsement* or *delegation*. These statements about a principal are a more useful basis for trust than names. However, names are useful as an add-on convenience to make the system easier to use. (See Section 8.2.)

The simplest kinds of endorsements are statements of fact about another principal. Here are two examples of slog/datalog facts used to represent user attributes in SAFE GENI.

```
geniUser($Alice).
geniPI($Alice).
```

For purposes of this example `$Alice` evaluates to the name of a unique principal (the user Alice). These statements endorse Alice to act in specified roles. In SAFE GENI, `geniUser` and `geniPI` are predicates whose meaning is that a given principal has registered as a GENI user and/or as a GENI Principal Investigator, has accepted the usage agreement, and has been approved by an authority.

It is easy for slang programs to compose such statements, group them in logic sets, and exchange them with other entities via SafeSets. (See Section 4.3.) Once exported the statements in the logic set are tagged with the name of the issuer and signed under its key: they are certified endorsements of the named principal by the issuer. Other principals may choose whether or not to accept an endorsement based on the identity of its issuer.

In SAFE GENI all logic statements name a principal by its key hash. In this example `$Alice` evaluates to the hash of Alice's public key. Slog itself does not rely on this convention: like datalog, it only requires that all principal names are unique; that is, they are distinct from the names of other principals. The key hash is a convenient principal name that is compact, globally unique, and secure. Moreover the public key name space is decentralized: it does not require any central authority to ensure these properties in the global name space. Although not required for trust, it is easy to add a symbolic namespace in SAFE GENI. (See Section 8.2.)

4.1 Subject Sets

A principal may pass its endorsements to another party as credentials to gain trust or access for a request. Principals in a networked system need some mechanism to collect and store endorsements and pass them to other parties. SAFE GENI uses a simple mechanism called a *subject set* for this purpose. Each principal runs the following slang code to construct and post a subject set.

```
defenv Self :- label("");

defcon makeMySubjectSet() :-
  {
     link($Self).
     label("subject($Self)").
  }.

defpost postSubjectSet() :-
  [makeMySubjectSet()].
```

In this code the value of `$Self` is the controlling principal's name. In this case it is the principal's key hash, which also serves as the token for the principal's identity certificate, as explained above. The `$Self` environment variable is defined in the `defenv` using the builtin `label` predicate as a function: used in this way, `label` returns a token for the set named by the given label string. The token corresponding to an empty

label is a hash of the principal's public key, which is given in `$SelfID`. (See Section 3.3.) This definition for the `$Self` environment variable is a standard convention of slang programs.

This code posts a new logic set whose token is derived from a string label identifying the subject by its key hash. This example shows the use of `$` to drive substitution in a quasi-quoted string. In a quoted string, a substring prefixed by `$` is interpreted as a variable—either a logic variable or an environment variable—and is substituted with the variable's value. The label value is a convention for subject sets: we may use any label string, as long as we are consistent about it.

Initially, the subject set contains only a link to the identity certificate, named by its token. The `link` statement adds the contents of the linked set (`$Self`) into the logic set by reference. Similarly, if some other principal issues an endorsement to this subject, the endorsee can link the new endorsement into its subject set, given a token for the endorsement. Here is the slang code for a principal to link a `Token` into its subject set:

```
defcon addTokenToSubjectSet(Token) :-
  spec("Update subject set to include a new token"),
  {
    link(Token).
    label("subject($Self)").
  }.

defpost updateSubjectSet(Token) :-
  [addTokenToSubjectSet(Token)].
```

The endorser must transmit the token for the new endorsement to the endorsee by some means. For example, it might send the token value in an RPC return, or place it in a web page or e-mail message for a user to cut-and-paste onto a command line. A token may be passed in on the command line as an environment variable.

The example illustrates automatic merging of logic sets based on the token or label. The `defcon` constructor `addTokenToSubjectSet` specifies the standard label for a subject set. If the subject set already exists, then the listed statements are added to that set, rather than constructing a new set. The `defcon` performs the read-modify-write on SafeSets automatically. One may also delete statements or links from a set by postfixing the statement with a tilde character. Note that it is generally not necessary for the read-modify-write to be atomic, since only one principal can modify any given set. However, if a principal controls multiple SAFE instances (e.g., a replicated server represents the principal) then we need some means to ensure consistency, e.g., the underlying key/value store must merge conflicting writes.

After collecting its endorsements in its subject set, a principal can pass them by reference to a server as credentials for an access request. To do this, it simply passes the token as a parameter to an RPC call. The server fetches and validates the subject set if it is not already present in its set cache. (See Section 5.)

4.2 Authority

The endorsements in the subject set are signed by their issuers. The authorizer considers them according to its policy rules. One role of policy rules is to represent who is trusted to make specific endorsements. Any principal may issue a statement endorsing the public key of another, but a faulty or malicious principal could make statements that are false. The policy rules enable an authorizer to determine whether or not to believe a given statement based on the identity of the issuer. These policy rules are also declared in slog logic. For example:

```
geniUser(User) :- identityProvider(IdP), IdP: geniUser(User).
geniPI(User) :- identityProvider(IdP), IdP: geniPI(User).
```

In these policy rules the terms `User` and `IdP` are logic variables representing principals. These rules specify a condition for a SAFE GENI server to accept that a given principal is a `geniUser` or a `geniPI`. The first goal of each rule `identityProvider(IdP)` is satisfied only if the value of the variable `IdP` names a principal that is believed locally to be a particular kind of authority called an identity provider. The second goal of each rule is satisfied only if the identity provider **says** (indicated in SAFE by the colon) that the User—whoever it is—is a registered GENI user or PI. This is an example of an *attribute-based delegation*: an authorizer believes that a given user is a GENI user or PI if any authorized identity provider says so. That is, the authorizer delegates to the identity providers the power to say who is a registered GENI user or PI.

Similarly, every SAFE GENI service uses a rule to determine if a given principal is an identity provider:

```
identityProvider(IdP) :- geniRoot(Geni), Geni: identityProvider(IdP).
```

This rule says that a principal is an identity provider if some other principal says that it is, where that principal is believed to be a `geniRoot`. In the GENI trust structure the GENI root is a principal that is accepted by members of a GENI federation to endorse identity providers and other authorities.

Every SAFE GENI service uses these rules as part of its standard policy. We presume that every GENI participant is configured with trust in one or more GENI roots: the service administrator may supply a `geniRoot` fact directly in the slang program, or the program may import it from a local file or posted set.

4.3 Issuing Endorsements

A principal issues an endorsement or delegation for another principal by posting statements in a logic set named by a token. The contents of the set are given in the endorser's slang program as a `defcon` with a template. For example, here is slang code for a GENI root to endorse an identity provider.

```
defcon idpEndorsement(IdP) :-
  {
    identityProvider(IdP).
    label("endorse($IdP)").
  }.

defpost endorseIdP(IdP) :-
    [idpEndorsement(IdP)].

defpost urlEndorseIdP(URL)
    [idpEndorsement(subject(@(URL))).
```

The endorser must have the principal name of the endorsee in the variable `IdP`. The name may be given on a command line or obtained by other means. Alternatively, this slang code (`urlEndorseIdP`) may use the `subject` and `@` builtins to harvest the endorsee's name from a secure URL, e.g., for a pem-encoded public key or identity certificate.

How does an endorser decide whether or not to issue an endorsement? In this example the trust is brokered out-of-band: it is based on the federation's procedures for qualifying its authority services, which may be operated by third parties.

Once the endorsement is issued, the identity provider must obtain its token by some means: for example, it may be posted on the Web, sent in an e-mail, returned in an RPC result, or written on a scrap of paper. Once the endorsee receives the endorsement token it adds the token to the credentials in its subject set. (See Section 4.1.)

Next, consider how the identity provider itself endorses users. An identity provider's authority to issue credentials flows from its own endorsement by a GENI root, as shown above. To substantiate the credentials it issues, the endorser may link its own credentials into the endorsement set. The slang code below endorses a principal as a `geniUser` and a `geniPI`, including a link to the issuer's own subject set. If the issuer is an authorized identity provider then its subject set contains its endorsement delegating that authority from the GENI root.

```
defenv SubjectSet :- label("subject($Self)").

defcon endorsePI(User) :-
  {
    geniUser(User).
    geniPI(User).
    link($SubjectSet).
    label("endorse($User)").
  }.

defpost geniPI(User) :-
  [endorsePI(User)]
```

The linked subject set makes the issuer's credentials available for inspection by anyone who receives the endorsements that it issues. This technique of linking each credential to the relevant credentials of its issuer is a fundamental design trick in SAFE. In particular, SAFE GENI constructs all credential sets in such a way that each set references all supporting credentials needed to validate it, as shown in more detail below. The set of linked supporting credentials for each logic set is called its *support set*. Linked support sets make it easy for an authorizer to obtain all credentials necessary for an authorization decision. (See Section 7.1.)

This example again raises the question: how would an identity provider know to issue this endorsement? SAFE GENI presumes that the identity provider is a Web service that receives endorsement requests from users, and bases its endorsements on external information about those users. In particular, users may authenticate with a password under a Web identity (SSO) protocol such as OAUTH or Shibboleth [11]. The identity provider issues credentials based on authenticated attributes bound to the user identity.

For example, GENI runs a portal service that harvests attributes about each academic user from a Shibboleth identity server at the user's institution. The institutional identity server is also called an "identity provider", so we might think of the GENI service as an "identity portal" to avoid confusion. Once logged into the portal with an institutional identity, the user may use Web forms to supply additional information to the portal and to accept required conditions. If the user provides its key hash, the portal may issue endorsements to approve the user as a GENI user and/or principal investigator (PI), based on attributes supplied by the institution via Shibboleth/ SAML (e.g., user is a faculty member).

5 Guards

The identity provider in Section 4.3 shows one way that GENI bootstraps trust from services outside of GENI. However, most GENI services perform access checks for requests based on credentials issued by GENI authorities. SAFE logic and slang enable us to specify these access checks (guards) directly in logic.

Slang supports guards via `defguard` rules. These rules define external entry points to the slang program for access checking of incoming requests. Each API method of the service has a corresponding guard in the slang program. SAFE integration support in the Web framework gathers parameters for the request and invokes the corresponding guard entry point.

The parameters are passed into the slang program as named environment variables. Several environment variables with predefined names are always present.

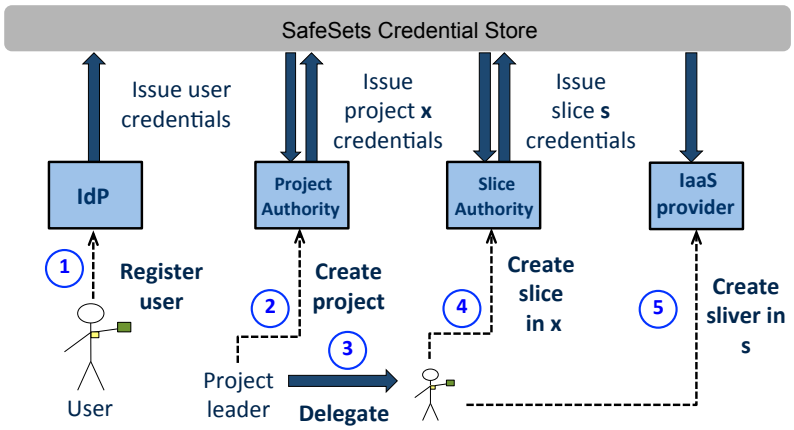

Figure 1: GENI authority services and credential flow for a typical resource request. Each virtual resource instance ("sliver") is obtained from a single IaaS resource provider ("aggregate"), and is bound to a project and slice accepted by that aggregate according to its policy. The authorities approve users, projects, and slices by issuing credentials. In SAFE GENI the downstream services fetch the credentials (logic sets) by reference from a shared SafeSets store and cache them locally.

- `$Subject` contains the authenticated principal name of the requester. It is the responsibility of the framework to authenticate the requester: one approach is to use secure HTTP as the transport and accept a client certificate.
- `$BearerRef` contains a single token passed by the requester, referencing 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 most other environment variables in that their values are specific to a request. Each request runs on a thread.

To see how the guard policies in a SAFE service use credentials and policy rules to check access, we consider the guards for the key GENI services. First, Section 5.1 summarizes the GENI abstractions and the GENI services that manage them, which are depicted in the canonical Figure 1. We then consider the guards to approve creation of three different GENI objects in the SAFE GENI example: projects (Section 5.2), slices, and slivers (Section 5.3). Sections 5.4 and 5.5 discuss the role of policy delegation in the example guards.

5.1 GENI in a Nutshell

A GENI deployment has a set of infrastructure hosting (IaaS) providers called *aggregates*. The tenants are GENI users, who may allocate and control virtual infrastructure elements spanning multiple aggregates, and link them together to form end-to-end environments (slices) for experiments or networked applications.

A *slice* is a logical container for a set of virtual infrastructure resources. The slice abstraction is useful to to name, control, and contain groups of virtual resources that span multiple provider sites and are allocated and used for a common purpose. A *sliver* is a virtual resource unit that is provisioned from a single aggregate and is named and managed independently of other slivers. Each sliver is bound to exactly one slice at the time that the sliver is created.

The privileges of a user in GENI to operate on slices and slivers depends in part on the user's membership in groups associated with specific activities. These groups are called *projects* in GENI. A Project Authority (PA) is an authority service that approves the creation of projects on behalf of users, who become the owners of the projects. The owner of a project may delegate membership privileges to other users.

In particular, a requester's right to create a new slice is determined by its membership in a project. Each slice is bound to exactly one project at the time the slice is created. The project and its leader are accountable for activity in the slice. Therefore, creating a slice imposes cost and risk on the project. Slice creation is permitted only for users wielding a specific named privilege (`instantiate`) within the project. A Slice Authority (SA) is an authority service that approves the creation of slices.

The GENI authorization framework uses slices as the granularity of access control for slivers. Any principal with privilege to control a slice may control its slivers. Aggregates determine those privileges based in part on statements by the SA that approved the slice. In particular, the SA names the owner of the slice and the project responsible for the slice. A capability-based access control model enables the owner of a slice to delegate control privileges to other users.

5.2 Projects

Here is the PA's guard condition to check whether a requester is authorized to create a project:

```
defguard createProject() :-
  spec("Guard for ProjectAuthority.createProject"),
  BaseRef := label("Local policy: GENI standard"),
  {
    link($BearerRef).
    link(BaseRef).
    geniPI($Subject)?
  }.
```

The guard is a rule prefixed with the `defguard` rule tag. A `defguard` rule captures the guard condition for the service API method of the same name: the `createProject` method.

The last goal of the guard rule defines a logic set, which is used as the context for the authorization query. The guard code selects the subsets to include in the context by linking to them in the usual fashion by their tokens. This context combines the requester's credentials and a named package of policy rules constructed during initialization of the server's slang program.

The context set for a guard invocation contains a query: any statement ending in a `?` is a query. Slang invokes the slog interpreter to issue the query against the other statements in the set, which comprise the context of the query. The returned value holds the result of the query.

In this case the guard condition is true if the authorizer "believes"—according to its policy rules and the subject's credentials—that the subject is a registered and approved GENI principal investigator (PI): the access policy for `createProject` is that any GENI PI can create a project. The policy rules in the context include the identityProvider rules of Section 4.2. The credentials in the context include the endorsements of Section 4.3. If these statements are valid and present in the context, then the PA's interpreter infers that the requester is an approved GENI PI.

5.3 Slices and Slivers

It is instructive to consider two more guards from the SAFE GENI example to understand more complex policies involving delegated rights to objects. We now consider the guard for creating a new slice, which runs at a Slice Authority (SA), and the guard for creating a sliver, which runs at an aggregate. These guards serve as examples for much of the rest of the paper.

To approve a slice, an SA must be convinced that the project is valid and approved by an eligible Project Authority (PA), and that the subject has permission within the project to bind a slice to it. Here is slang code for a standard guard for the `createSlice` method.

```
defcon sliceAuthorityPolicy() :-
  {
    approveSlice(Project, Owner) :-
      PA := rootPrincipal(Project),
      projectAuthority(PA),
      PA: project(Project, standard),
      PA: memberPrivilege(Owner, Project, instantiate),
      geniUser(Owner).

    label("Slice policy: create; Type: standard").
  }.

defguard createSlice() :-
  CreatePolicyRef := label("Slice policy: create; Type: standard"),
  BasePolicyRef := label("Slice policy: base; Type: standard"),
  {
    link($BearerRef).
    link(BasePolicyRef).
    link(CreatePolicyRef).
    approveSlice($Object, $Subject)?
  }.
```

The `createSlice` guard constructs a query context for the guard condition in the same way as the previous example, by linking a named set of policy rules and the credentials passed by the subject. The guard condition queries a predicate called `approveSlice`, which captures the preconditions for the SA to permit a new slice with the requesting subject as its owner.

In this case the target object is the project for the requested slice, whose name is passed in `$Object`. There are two basic conditions to approve the new slice. First, the SA must believe that the project is valid. A project is valid if an authorized Project Authority (PA) has issued a valid statement that the project is a GENI project. Second, the SA must believe that the requester is an approved `geniUser` wielding `instantiate` privilege within the project.

To validate these conditions, the guard first uses the `rootPrincipal` builtin to obtain the name of the principal that created the object. (See below.) It then checks that the root principal is a valid Project Authority, and checks that the PA asserts that the object is a valid GENI project. It also checks that the PA **says** that the requester has `instantiate` privilege in the project. In effect this SAFE GENI guard delegates power to the PA to issue the rules governing use of each project. (See below.) Finally, the SA checks that the user is an approved `geniUser`.

The SAFE GENI infrastructure providers (aggregates) use similar guard and policy rules for sliver creation:

```
approveSliver(Slice, Owner) :-
  SA := rootPrincipal(Slice),
  SA: slice(Slice, Project, standard),
  sliceAuthority(SA),
  SA: controlPrivilege(Owner, Slice, instantiate, _),
  geniUser(Owner).

defguard createSliver() :-
  BaseRef := label("Aggregate policy: slice operations"),
  {
    link($BearerRef).
    link(...).
    approveSliver($Object, $Subject)?
```

```
}.
```

This guard verifies that the target object is a slice and that the slice is valid: an authorized Slice Authority (SA) has issued a valid statement that it is a GENI slice bound to an active GENI project of some class ("standard"). It also checks that the requester is an approved `geniUser` wielding `instantiate` permission on the slice.

5.4 Policy Delegation

The `createSlice` and `createSliver` examples illustrate the flexibility of guards in SAFE. In particular, they show how a service may delegate the policy rules for the privilege check on the target object to another principal. These guards delegate policy control over the target object to its *root principal* as returned by `rootPrincipal` builtin. For example, the root principal for a slice is the Slice Authority who approved creation of the slice, assigned it a name, and endorsed it. Section 6 discusses object naming and authority.

This policy delegation relies on mobility of policy rules, and the inherent ability of logical trust systems to reason from policy rules issued by other principals. An imported policy rule enables an authorizer to infer beliefs of the issuer, given some set of facts. For example, the `memberPrivilege` goal in a SAFE GENI Slice Authority's `createSlice` guard is satisfied if the PA itself has issued policy rules for `memberPrivilege` for the project, and the goal can be inferred from those rules and the credentials passed by the requester. In essence, the **says** operator in a goal may be understood as "says or believes" and the SA's guard requirement is that "the PA would believe it if it saw what I see". In the case of an aggregate's `createSliver` guard, the policy control rests with the Slice Authority that endorses the slice: the goal requires that the SA "says or believes" that the user has the permission. The PA and SA may define the policy rules governing access for each project and each slice, and may specify different rules for different objects.

It is easy to pass policy rules in SAFE and apply imported policy rules safely. Any posted logic set may contain policy rules. On fetch, an authorizer verifies that the head of any received statement—whether or not it is a rule—is tagged with its authenticated issuer (using **says**) before accepting it. Once the imported rule is added to a context, the engine applies it only to infer beliefs of its issuer, according to the **says** tag on the rule's head. It cannot affect any policy decision unless some goal condition explicitly delegates to the issuer by naming it, as in these examples.

The `createSliver` guard illustrates a related form of policy delegation. If an aggregate accepts an SA as a Slice Authority according to the rules of its federation then it accepts any slice approved by that SA. This guard validates neither the project bound to the slice nor the PA that approved the project. Instead, it trusts the SA to do that.

The author of a guard may delegate policy control in this way or choose to impose local policy restrictions on either the subject or object, or on other principals or objects related to the request. For example, it is a simple matter for an aggregate to validate the project in the `createSliver` guard, or to apply other local restrictions on the projects accepted at this aggregate. Similarly, the guard could enforce any local policy restrictions on the identity of the user. This guard validates that the user is a valid `geniUser` according to its own policy, independent of any policy of the SA.

5.5 Trust Structure

With these examples of guards and policy delegation in hand, we are now ready to discuss the structure of a federation, such as a GENI deployment.

Consider how the SAFE GENI servers validate that a principal is an "authorized" Project Authority or Slice Authority in these guards. Any server could impose its own conditions to satisfy the `projectAuthority(PA)` and `sliceAuthority(SA)` goals in Section 5.3 or the `identityProvider(IdP)` goal in Section 4.2. But in

these examples the servers delegate this control to the GENI federation root(s). They do this by including rules like these in their policies:

```
projectAuthority(PA) :- geniRoot(Geni), Geni: projectAuthority(PA).
sliceAuthority(SA) :- geniRoot(Geni), Geni: sliceAuthority(SA).
```

These rules delegate to a GENI root (`Geni`) the power to say who is an Authority in the federation, and what authority roles are allowed to each server. They state that a principal is an Authority if the federation root says or believes that it is. They are similar to the identity provider rule discussed in Section 4.2.

To satisfy these rules, the root might simply issue direct endorsements of the federation's SA and PA servers, following the Identity Provider example in Section 4.3. An authority server includes any endorsements from the federation root in its subject set, and attaches it to the credentials it issues for inspection downstream. That is sufficient to implement a simple SAFE GENI federation.

As the federation evolves, the root may define additional rules to implement changes to the trust structure. These rules may be incorporated into the policies of other federation members and applied automatically, without changing their code. In particular, the root may add rules to peer with another federation according to various terms and policies. Section 7.3 discusses some example alternatives.

6 Objects

Slices and projects are "global objects" in GENI. What makes them global is that multiple servers make access control decisions regarding these objects. For example, Section 5.3 shows how a Slice Authority requires a requester to have specific access rights for a project in order to create a GENI slice in the project. The Slice Authority bases its choice entirely on statements about the project by third parties, including the project's root principal (a Project Authority) and its delegates. Similarly, a requester's right to create or manipulate a sliver at an aggregate is determined solely by the privileges that it holds over the containing slice.

More generally, an "object" could be any element we wish to describe with the logic. The set of objects and the lifetimes of objects are dynamic: objects come and go. Entities may create objects and assign names to those objects without approval of any central naming authority. Entities may issue statements containing any object names they choose. SAFE imposes a convention for object names, but the slang programs may define and use arbitrary predicates over the objects.

6.1 Object Naming and Authority

Statements controlling access to an object are authoritative only if they are issued by an object's root principal or its delegates. Section 5.3 shows how guards use the `rootPrincipal` builtin to verify this.

The `rootPrincipal` builtin relies on a naming convention for objects to extract and return the root principal name securely. Like principal names, object names must be unique and distinct strings in order for logical inference to be sound. The names should also be authenticated, so that principals cannot "hijack" one another's objects by issuing unauthorized statements about them. To ensure these properties, SAFE defines a convention for global object names. A fully qualified global object name is a *universally/globally unique identifier* (UUID/GUID) concatenated with the name of the object's root principal—the principal that assigns the name and **controls** the object. We refer to the fully qualified name as a *self-certifying identifier (scid)*. In SAFE GENI all logic statements about global objects name them with scids.

The following slang code creates a new global object and publishes a logic set describing it. We refer to a logic set describing an object as an *object credential*. Every GENI object (project, slice, sliver) has an object credential in SAFE GENI. The object credential is issued by the object's root principal. It contains

attributes of the object and binds the object to a principal who owns the object or (optionally) is not the owner but is authorized by the root principal to operate on the object.

The SAFE GENI Project Authority uses this code to register a new project and issue an object credential—in this case a project credential:

```
defcon standardProject() :-
  spec("Endorse a standard project"),
  StdPolicy := label("object(standardProjectMembership)"),
  Project := scid(),
  {
    owner($Subject, Project).
    project(Project).
    link($SubjectSet).
    link(StdPolicy).

    label("project credential ($Project, $Subject)").
  }.

defpost createProject() :- [standardProject()].
```

A Project Authority invokes this code after processing an approved `createProject` request. (See Section 5.2.) The `scid()` builtin mints a 128-bit GUID according to IETF RFC 4122, and returns a self-certifying object identifier for the new object based on the GUID and the controlling principal name in `$Self`. The remaining code uses slang features discussed in earlier sections. It posts a logic set—a project credential—that links to the issuer's subject set and a package of standard policy rules constructed during initialization.

In this case, the posted logic set includes two facts about the new `Project`. The statement `project(Project)` indicates that the object is a project, as required by the `createSlice` guard in Section 5.3. The statement `owner($Subject, Project)` indicates that the requester is the "owner" of the new project.

A Slice Authority uses similar slang code to issue a slice credential for a newly created slice:

```
defcon standardSlice(Project) :-
  spec("Endorse a standard slice"),
  StandardSliceOps := label("object(standardSliceControl)"),
  StandardSliceOpsDefault := label("object(standardSliceDefaultPrivilege)"),
  Slice := scid(),
  {
    owner($Subject, Slice).
    slice(Slice, Project, standard).

    link($SubjectSet).
    link(StandardSliceOps).
    link(StandardSliceOpsDefault).

    label("slice credential ($Slice, $Subject)") :-
  }.

defpost createSlice() :- [standardSlice($Object)].
```

The posted logic set is a slice credential. Like the project example, it declares the requester (subject) as the owner of the slice. This example includes more information about the slice: it declares the project that the slice is bound to and a class for the slice ("standard"). Guard policy rules may use this information to

distinguish different classes of slices. The slice credential also links to two policy rule sets governing access to the slice. Any authorizer evaluating this slice credential imports the issuer's policy rules for the slice automatically. Sections 6.2 and 6.3 consider these policy rules.

6.2 Object Delegation

Section 5.4 discusses how guards may delegate control over access policy to an object's root principal. In the case of projects and slices, the root principal is an authority within the GENI federation. Section 6.1 shows how the authorities post credentials for these objects, with links to the policy rule sets. This section considers the policy rules.

The policy rule set for the project credential is given by the variable `StdPolicy`. These rules delegate control over membership in the project to its owner. They permit the owner to delegate membership to other principals. Optionally, the delegation may be transitive. Here is a snippet of a slang `defcon` to construct the policy set:

```
defcon standardProjectMembership() :-
  spec("Membership policy for a standard project"),
  {
    member(User, Project, true) :-
      owner(User, Project).

    member(User, Project, Delegatable) :-
      Delegator: delegateMember(User, Project, Delegatable),
      member(Delegator, Project, true).

      ...

    label("object(standardProjectMembership)").
  }.
```

The first rule in the constructed policy set states that the owner of a project is a member. The `true` in the third parameter indicates that the membership is delegatable. The second rule defines how an authorizer should interpret a delegation: if a principal `Delegator` with a delegatable membership says that another principal `User` is a member of the project, then it is so. The delegated membership may itself be delegatable, or not, at the option of the delegator.

The first rule set for slices is similar:

```
defcon standardSliceControl() :-
  spec("Control policy for a standard slice"),
  {
    controls(Subject, Slice, true) :-
      owner(Subject, Slice).

    controls(Subject, Slice, Delegatable) :-
      Delegator: delegateControl(Subject, Slice, Delegatable),
      controls(Delegator, Slice, true).

    ...

    label("object(standardSliceControl)").
  }.
```

These rules allow a project owner or slice owner to delegate membership or control to another user. Issuing a delegation is no different from issuing any kind of endorsement. (See Section 4.3.) A principal issues a delegation by posting it as a logic set named by a token. For example, it may run a simple slang program from a command line:

```
defcon delegateMembership(User, Project, Delegatable) :-
   {
       delegateMember(User, Project, Delegatable).
       link($SubjectSet).
       label("delegate($User, $Project)").
  }.

defpost postDelegation(User, Project, Delegatable) :-
    [delegateMembership(User, Project, Delegatable)].

definit issueDelegation() :-
    postDelegation($To, $For, true).
```

The delegator must have the principal name of the endorsee, i.e., its key hash, and the name of the project. This code presumes they are passed on the command line.

Once the delegation is issued, the receiver must obtain its token by some means. One simple scenario is that the issuer cuts-and-pastes the generated set token into an e-mail. The receiver then runs the `addTokenToSubjectSet` script to link the token into its subject set. A user may run this script as a command, pasting the token onto the command line. (See Section 4.1.)

The object delegation example reinforces a core principle of SAFE. Each credential, endorsement, or delegation includes one or more links to logic sets that contain the credentials demonstrating the issuer's authority to issue that delegation. We refer to these sets as *support sets*. If all participants follow the rule, then each user's token leads (transitively) to the full set of credentials needed to demonstrate access. The receiver constructs its own support sets by adding each delegation to a set as it receives it. Section 7.1 discusses this topic in more detail.

6.3 Capabilities and Refinement

The object delegations discussed in Section 6.2 are a form of capability-based access control, specified and implemented in trust logic. As specified above, it supports *confinement*, the ability of a delegator to block the receiver from delegating the privilege to a third party ("the friend of my friend is my friend"). We now show how to expand the capability support for *refinement*: the ability of a delegator to constrain the access privileges permitted under a delegation. Refinement is an important element of capability models. To support refinement, the project policy set adds these rules:

```
memberPrivilege(User, Project, instantiate, Delegatable) :-
  member(User, Project, Delegatable).

memberPrivilege(User, Project, info, Delegatable) :-
  member(User, Project, Delegatable).

memberPrivilege(User, Project, Priv, Delegatable) :-
  Delegator: delegateMemberPrivilege(User, Project, Priv, Delegatable),
  memberPrivilege(Delegator, Project, Priv, true).
```

This example declares two distinct privileges for a project: `instantiate` and `info`. The `instantiate` privilege represents a right to instantiate slices within the project. The `createSlice` guard in Section 5.3

requires this privilege. The `info` privilege represents a right to request information about the project. These privilege tags are simply string constants: we can have as few of them or as many of them as we need, and we can call them whatever we want, as long as we use the names consistently.

The first two rules confer `instantiate` and `info` privilege on any full-fledged member of the project. The third rule permits a holder to delegate either privilege independently of the other. The receiver of a `delegateMemberPrivilege` delegation is not a full-fledged member of the project; instead, the receiver has only the delegated privilege. The delegated privilege may itself be delegatable, or not.

Capability protection for GENI slices (SFA capability model) is represented with similar rules. This example changes the privilege types and shortens the `Delegatable` variable:

```
controlPrivilege(Subject, Slice, instantiate, Dlg):- controls(Subject, Slice, Dlg).
controlPrivilege(Subject, Slice, info, Dlg) :- controls(Subject, Slice, Dlg).
controlPrivilege(Subject, Slice, start, Dlg) :- controls(Subject, Slice, Dlg).
controlPrivilege(Subject, Slice, stop, Dlg) :- controls(Subject, Slice, Dlg).

controlPrivilege(Subject, Slice, Priv, Delegatable) :-
  Delegator: delegateControlPrivilege(Subject, Slice, Priv, Delegatable),
  controlPrivilege(Delegator, Slice, Priv, true).
```

The holder of a delegatable capability may delegate by issuing a credential (a logic set) as described in Section 6.2. The credential incorporates a `memberPrivilege` or `controlPrivilege` assertion instead of a statement of full membership or control.

One advantage of implementing capability-based access control in trust logic is is that it is easy for a server to impose additional conditions for access, according to its own policies ("gelded capabilities"). The guards in Section 5.3 illustrate this flexibility: they deny access to a user who is not a registered `geniUser`, even if the user wields a valid capability. Section 7.2 presents further examples of variations to the capability model that are useful in GENI and are easily realized in trust logic.

6.4 Speaks-For

The guards shown here do not show support for the **speaksFor** operator. GENI uses **speaksFor** to allow a Web-based user portal to issue requests on behalf of users, with the user's permission.

SAFE handles **speaksFor** transparently. A **speaksFor** authorization is an ordinary logic statement. For example, a user issues a statement endorsing a portal to speak for the user. SAFE maintains a cache of **speaksFor** authorizations it has encountered. If it encounters any request or credential whose authenticated issuer does not match the named speaker, then it rejects it unless a matching **speaksFor** authorization is present. If an authorization is present, it passes the content into the logic system as if it had been issued by the named speaker.

7 Set Linking in SAFE GENI

SAFE GENI illustrates how trust logic can work in practice in a networked system. An important requirement is that all participants must agree on the meaning of the predicate symbols and names used in credentials. One way to approach this is to formulate a standard for the vocabulary, publish it, and develop tests to ensure that all participants comply with the standard.

In SAFE GENI we instead presume that all participants run the same slang program, which implements the core authorization model in a few hundred lines. It is straightforward to do this because SAFE runs as a separate process, and is therefore decoupled from the development environment at any given site or service.

The slang program embodies all that the participants need to know about the logic and the GENI vocabulary. The SAFE process is stateless, so participants may restart it and/or reload the slang program at any time: it affects only the access control for future requests. Changing the program leaves other software and state unchanged at the site.

Moreover, much of the system behavior is determined not by the program itself but by the policy rules in effect at each authorizer. Any participant may add local rules to tailor the policies to local needs, without changing the slang program. Slang programs are composable: it is easy to load another local program with a `definit` rule that adds slog rules to the standard policy packages with known labels. Participants may even formulate rules and exchange them over the network as the system executes.

SAFE supports these properties in part through the uniform access to logic sets in a shared store (SafeSets). Each authorizer fetches referenced credentials on demand, validates them once, and caches their logic content. The slang runtime handles the details of exporting and importing logic sets automatically. This approach to credential management is simpler, faster, and more flexible than the common approach of passing credentials by value with every request.

Set tokens also enable *set linking*, a powerful technique to organize the logic content—credentials and policy rules—produced by the participants in a networked system. Any set constructor may use the `link` predicate to link to another set by reference, given its token. SAFE GENI makes extensive use of set linking to build query contexts, so that an authorizer can retrieve all credentials that are relevant to any given authorization decision easily (Section 7.1). The cost of linking is low because common subsets are cached at each authorizer. SAFE GENI also uses set linking for policy sets to enable flexible control over the access rules in effect for objects (Section 7.2) and a dynamically evolving federation trust structure (Section 7.3).

7.1 Set Linking and Support Sets

Set linking in SAFE GENI naturally organizes credentials into a graph that reflects the delegation structure. Our approach relies on decentralized construction of the credential graph. The issuer of any credential—a delegation or endorsement—links the issued logic set to any of its own credentials that support its authority to do so. Each credential is materialized as a certificate containing one or more links to additional certificates, each of which may also contain links. If all issuers follow this convention then the transitive closure of any given certificate 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 slog inference engine to validate that these chains lead back to one or more trust anchors (e.g., a `geniRoot`) according to its policies.

This "inductive construction" or "constructive induction" approach is SAFE's solution to the *credential discovery problem*, long recognized as a fundamental challenge for logical trust systems. Previous approaches in the literature suggest various on-demand query models in which an authorizer retrieves relevant credentials by executing a distributed query for statements whose semantic content matches goal statements of the authorization procedure (e.g., [10, 3]). We believe that the constructive approach is simple and it avoids the need for a distributed query infrastructure. However, it relies on advance knowledge about how an authorizer ultimately uses the credentials. The SAFE GENI example illustrates that this approach is practical even in a relatively complex application scenario.

To illustrate, consider how a principal uses an object delegation (e.g., a capability) to access an object. It stores the delegation in its subject set and passes the subject set token as the `$BearerRef` in the RPC call to a server. The server's guard fetches the set by linking it into its context for the guard query. (See Section 5.3.) The object delegation links to the subject set of its issuer, which similarly contains credentials validating the issuer's control over the object. The context therefore includes (transitively) the entire delegation chain leading back to the root object credential. This completeness condition is sufficient for the guard query to validate access.

In particular, the object credential includes a link to the root principal's subject set, which contains the

endorsements that qualify it as an authority to approve slices or projects according to the rules of the federation. (See Sections 5.5 and 6.1.) We emphasize that in practice, a server finds frequently linked supporting credentials in its set cache, and does not fetch or validate them again on each request. For example, every object credential issued by a given Slice Authority links to its own credentials (in its subject set), but a GENI server operating on many slices maintains only one copy of the SA's credentials in its cache, and it only validates them on first fetch. If an authorizer receives a slice credential issued by a new SA, it automatically validates its credentials and fetches them into its set cache.

The SAFE GENI example presumes for simplicity that all participants use their subject sets to track their credentials. One downside of this choice is that a principal with a large number of credentials ends up passing everything to everybody. This might not be the best choice on grounds of confidentiality and performance. But every principal is free to organize its credentials as it sees fit: a user may arbitrarily spread its endorsements and delegations across multiple sets. For example, it might use a separate set for each project and for each slice. What is important is that each issuer links the correct support set(s) into each endorsement or delegation that it issues, and passes the correct tokens to justify each request.

7.2 Hybrid Access Control

Another advantage of set linking is that it naturally supports policy mobility. In particular, the linked logic sets may include policy rules. The receiver applies the rules automatically: if the receiver incorporates an imported rule into a query context, the inference engine uses the rule. It is safe to apply the rules: the inference engine uses any rule only to the extent that other policies of the authorizer delegate policy control to the issuer of the rule. (See Section 5.4.)

For example, set linking enables an object's root principal to control the terms of the object's use by linking policy rules into an issued object credential. (See Section 6.1.) The root principal may vary these terms from one object to another or change them at any time.

The flexibility of trust logic and set linking enables useful additions to the capability-based access control model on a per-object basis. This flexibility is useful in SAFE GENI: Slice Authorities add policy rules to enable additional paths to gain access outside of the capability model. The SAFE GENI example includes four exemplary add-on rules for slices:

```
defcon standardSliceDefaultPriv() :-
  desc("Default privileges for standard slices"),
  {
    controlPrivilege(Subject, Slice, info, Delegatable) :-
      slice(Slice, Project, standard),
      PA := rootSubject(Project),
      projectAuthority(PA),
      PA: project(Project),
      PA: memberPrivilege(Subject, Project, info, Delegatable).

    controlPrivilege(Subject, Slice, stop, true) :-
      slice(Slice, Project, standard),
      PA := rootSubject(Project),
      projectAuthority(PA),
      PA: project(Project),
      owner(Subject, Project).

    controlPrivilege(Subject, Slice, info) :-
      slice(Slice, Project, standard),
      gmoc(Subject).
```

```
  controlPrivilege(Subject, Slice, stop) :-
    slice(Slice, Project, standard),
    gmoc(Subject).

  label("object(standardSliceDefaultPrivilege)").
}.
```

The first rule enables any project member to obtain any information about any slice associated with the project. The second rule grants the leader of a project permission to query or stop any slice associated with the project.

The last two rules grant query/stop privilege over all "standard" slices to any principal accepted locally as a GMOC—GENI Management and Operations Center. These rules implement the GENI "kill switch". It is a simple matter for a Slice Authority to disable the kill switch on a per-slice basis (e.g., for non-GENI slices) by omitting the rules that enable it.

7.3 Federation Policy and Peering

Section 5.5 shows how a federation root can issue delegations and endorsements that define the federation trust structure, including the set of approved authorities. Additionally, the natural mobility of policy rules makes it easy for the federation root to issue policies for the federation. The federation members (voluntarily) fetch the federation policy rules and apply them directly. The root may change the policy rules over time. For example, as the federation evolves, a root may issue new rules to define changes to the federation trust structure. In particular, it may issue rules that define the terms of peering with a partner federation.

To publish policy rules, the GENI root posts them to a logic set with a standard label known to its federation members. The members link the root's policy set into every context for a guard query. For example, suppose the root links the policy rules into its subject set. (See Section 4.1.) Then each SAFE GENI server can import it by inserting the following link into every context, e.g., by including it in the server's base policy set:

```
RootRef := label("subject($GENI)", $GENI).
{
   link(RootRef).
}
```

This example presumes without loss of generality that each server is a member of a single local federation, whose root principal is given in the environment variable `$GENI`. It uses a form of the `label` builtin function that takes a principal name as a separate parameter; this variant synthesizes the token of a logic set belonging to another principal, given knowledge of that principal's key hash and the string label that it uses for the target set. In this case, the conventions are that the root includes the desired rules in its subject set, and all entities in the system use a standard `label` template for the subject set.

Now, suppose that the federation root wants to peer with another SAFE GENI federation whose root is given by the environment variable (`$Peer`). To accept users from the peer federation, it could simply add the following policy rule to its subject set:

```
identityProvider(IdP) :- $Peer: identityProvider(IdP).
```

With the addition of this rule (and a link to the peer's subject set), all services in the local federation accept user endorsements from identity providers in the peer federation. It enables the peer's users to join projects and access slices, subject to additional rules for membership and access that require consent of the project leaders and slice owners. (See Section 6.3.) The root could go further by adding a similar rule to accept the peer's Project Authorities, enabling members of the peer's projects to create slices and request resources for

them even without consent of local project leaders. Accepting the peer's Slice Authorities enables slices from projects in the peer federation to request resources from local aggregates.

Accepting the peer's authorities in this way is a simple option for cross-federation, but it is permissive in that it accepts guest principals on an equal basis with members in the local federation. For example, accepting a peer's identity providers enables PIs from the peer to create projects, according to the example `createProject` guard policy in Section 5.2.

It may instead be preferable to distinguish users, slices, and/or projects originating from the peer. A less permissive alternative is to accept the peer's users and/or projects and slices on a guest basis. For example, consider these root policy rules:

```
guestUser(User) :- IdP: geniUser(User), $Peer: identityProvider(IdP).
geniUser(User) :- guestUser(User).
```

These rules enable peer users to act with the same privileges as local federation users, according to the SAFE GENI example guards discussed earlier. But the rules confer no PI privilege to the peer's PIs, so they cannot, for example, create projects in the local federation.

Alternatively, if the second rule is deleted, then guest users have no access privilege unless other rules affirmatively grant it. In this way the guards may limit the scope of their privilege.

Note also that the first rule enables an authorizer to distinguish a guest user. It is another question how the authorizer's guard could deny access on that basis. We intend that SAFE will support a limited form of negation via *deny conditions* (blacklists) in `defguard` rules. A deny condition is a guard query that results in denying access if it evaluates to true. (Deny conditions are not yet supported in our prototype.) Deny conditions illustrate the power of splitting a logical trust system into a scripting layer (slang) and a core trust logic (slog): they are easy to support at the slang layer without any support for negation in the datalog core.

A third option for cross-federation peering is for individual authorities or aggregates to accept upstream authorities from a peer federation. For example, an aggregate may simply accept multiple federation roots, and thereby grant access to all of the users, slices, and projects of multiple federations on an equal basis.

It is an open question whether this permissiveness would be allowed under the rules of any individual federation. That is, by accepting foreign users, the aggregate may be violating its agreement with a sponsoring federation. The nature of the agreements that govern a priori trust relationships are outside the scope of SAFE. For example, SAFE can represent the trust policies to implement an aggregate's usage agreement with a sponsoring federation, but it cannot represent the agreement itself or force the aggregate to use only policies that comply with its agreement.

8 SAFE GENI and Real GENI

SAFE GENI implements the GENI authorization architecture in a few hundred lines of code. In particular, the various authority services (IdP, PA, SA) do little more than produce and consume credentials, so they can be implemented almost entirely in SAFE. For example, the basic function of the Project Authority is to check access for `createProject` and then post and return the new project credential. The example SAFE code shows how to do this: all that is missing is the Web API for a client to invoke the service.

As noted, SAFE GENI is not compatible with the GENI implementations deployed at the time of this writing. For various practical and technical reasons the GENI Project Office backed intermediate standards and implementations that do not use trust logic. These standards are now well-established in the GENI community.

This section comments on other GENI standards used in the NSF GENI deployment, their relationship to a pure logic-based implementation, and the potential for a more complete and/or interoperable GENI

implementation using SAFE. The underlying research question is: how far can we go with using logic to build a system like GENI? If we implement as much of the system as we profitably can using declarative logic, what remains? Why do we need it?

8.1 GENI Credentials

GENI defines custom credential (certificate) formats to transport assertions about GENI principals and objects. Principals and objects have symbolic names from a URN namespace defined for GENI principals and objects. GENI certificates name entities by their URNs and are authenticated by digital signature in the usual way. Some of the semantic content of a GENI credential is encoded in the URNs. Sections 8.2 and 8.3 discuss URNs.

GENI uses x.509 identity certificates to bind principal URNs to public keys, and a separate credential format for **speaks-for** assertions. GENI also defines an object credential format containing the object's URN and various essential attributes of the object—or the rights that some principal has for the object—depending on the type of the object and certificate. A related GENI standard defines a certificate format to delegate named rights over objects, based on the SFA capability standard.

SAFE GENI uses a general SAFE certificate format as a transport for signed logic. The logic payload is ordinary text written in the slog language. Therefore, the content of the logic payload in a certificate is flexible: we can make arbitrary changes to the content of the certificates without making any changes to the certificate standard or to the encoding/decoding software. The SAFE inference engine is similarly general-purpose: it can process any slog logic content.

This approach is more flexible and evolvable than the credentials used in the NSF GENI deployment. For example, slang programs can introduce their own vocabulary of logic predicates without changing any other software or certificate formats. The vocabularies of different participants cannot conflict unless they delegate trust to one another, in which case they must coordinate the vocabulary.

The updated GENI protocol standards (AM API 3.0) are specified to be open to alternative credential formats, so the GENI standards could evolve to accept SAFE certificates in the future. In any case, the semantic content of any GENI certificate is easy to express in slog. That introduces the possibility of adding a decoder plugin for SAFE GENI that imports GENI certificates into slog. The inference engine can reason from these certificates using declarative policy as described in this document. In this way, SAFE GENI aggregates might serve users with existing GENI tooling, and we can retrofit declarative policy to existing GENI aggregates.

8.2 Naming and GENI URNs

GENI defines a structured symbolic name space (URNs) for GENI objects and principals. Each entity in GENI is associated with a single domain: the entity's URN has a prefix that is the name of its containing domain. The domain name is presumed to be globally unique and stable: in practice a DNS name is used. The URN for an object or principal incorporates a user-assigned *common name* that is unique within its containing domain.

GENI uses an ordinary X.509 PKI hierarchy to bind public keys to principal URNs. A root Certifying Authority (`ch.geni.net`) issues X.509 identity certificates endorsing public keys for the domains; the domains then issue certificates endorsing public keys for their object authorities and other principals. The distinguishing name in a GENI identity certificate is a GENI URN.

Trust statements in SAFE GENI do not use symbolic names, but instead use machine-generated names directly. We may call them *cryptographic names* or *identifiers* because they incorporate the controlling principal's public key in such a way that the owner of any name can be authenticated cryptographically. As we have seen, in SAFE GENI a principal's identifier is the hash of its public key; an object identifier

combines a standard UUID/GUID (RFC 4122) with the public key hash of its controlling authority. Tokens are also cryptographic names: a logic set token is the hash of the issuer's public key combined with specific other information according to the label template.

We have not implemented or discussed symbolic names for SAFE GENI because they are not a necessary part of the trust system. Symbolic names (e.g., URNs) are merely a convenient add-on for a trust structure based on cryptographic names. The SPKI/SDSI RFC [7] has a detailed discussion on this point, which we now apply to the GENI example.

The primary virtue of symbolic names is that they are readable by humans, while cryptographic identifiers are unreadable sequences of random characters. Symbolic names help with user interactions such as Web forms, command line interfaces, and event logs. But they do not replace cryptographic names: a secure system requires cryptographic names for authentication.

A second benefit of symbolic names is that they offer a stable level of indirection to decouple identity from keypairs. A principal P may have a symbolic name that is bound to multiple keypairs, possibly at different times. Other principals may make statements about P that reference P by its symbolic name. These statements are true for any keypair that P uses, so long as some valid identity certificate binds the keypair's public key to P's name. The statements continue to be valid even if P rotates its keypair or uses multiple keypairs at once.

However, the price of this convenience is a dependency on a central naming authority to ensure that symbolic names are globally unique and stable. As noted above, the URN system for the NSF GENI deployment presumes such a global naming root: `ch.geni.net`. A future cross-federation of GENI deployments might rely on the existing Internet DNS system for global names. SAFE GENI shows that dependency on a global naming root is not a necessary part of the architecture.

One problem with symbolic names is that they are not guaranteed to be unique through time: users may assign the same common name to different objects at different times. The AM API 3.0 standard resolves this problem by adding a unique machine-generated identifier (a UUID/GUID) for each object; certain request APIs now use the URN and the UUID together to assure uniqueness.

8.3 Adding URNs to SAFE GENI

We can capture the GENI URN constructs in SAFE logic, given a clear standard for the syntax and encoding of GENI URNs. This extension to SAFE GENI is left as an exercise, at least for now, but here is a rough sketch. First, we use simple logic statements to represent the identity statements in GENI identity certificates. These statements are easy to represent: they merely bind principals and objects to URN strings, naming the principals and objects by their identifiers in the usual fashion. Second, we add rules to capture the indirection principle: any statement made about a URN is accepted as true for any identifier with a valid binding to that URN. For example, a statement that names a principal by its URN is true for any public key K for which a valid identity statement/certificate binds K to the URN. Third, we add validation rules for the identity statements themselves.

These validation rules require new builtin predicates to verify the hierarchical relationships of URNs in accordance with the syntax of the URN name space. An identity statement for a URN is accepted only if its speaker is the parent in the URN name space, i.e., the certificate is signed under a public key of the parent. The syntactic constraints on URNs (summarized above) enable this validation. Given the URN of a principal one can derive syntactically the URN of its parent in the identity hierarchy. Given the URN of an object, one can derive the URN of its controlling authority. We need two builtin predicates to validate pairs of URNs for a syntactic match, i.e., to verify that the issuer of a GENI identity certificate or object certificate is accepted as authoritative for the URN named in the certificate.[2]

[2]The archived `dev@geni.net` exchange of 8/5/14 is instructive. We note in passing that a stated motivation for the GENI URN standard was the need to defend against name hijacking and collisions by qualifying the object name with its controlling domain. However, cryptographic identifiers already have this property: it was the purpose of the *self-certifying object identifier* proposed in the SFA 2.0 document in April of 2010. Given a SAFE GENI object name one can extract the name (key hash) of

Finally, we need a few builtin predicates to extract other information encoded in URNs. In particular, object URNs encode the type of the object and essential relationships among objects. For example, the URN of a slice encodes the name of its containing project, and the URN of a sliver encodes the name of its containing slice. SAFE GENI represents these relationships as logic facts available to the inference engine. But it is easy to infer them from the URNs.

8.4 Using URLs as Symbolic Names

A simple alternative for user-readable symbolic names is to name objects with URLs structured similarly to GENI URNs. Suppose that each principal controls a web directory corresponding to its symbolic principal name following the URN standard (e.g., `exogeni.net/sa` or an ordinary user web directory. To install a nickname for an object (e.g., `myslice`), the principal adds a file with that name to its directory. The content of the file is the corresponding cryptographic identifier. Programs that handle input from a user may use the SAFE `@` operator to fetch the contents of the URL and obtain the underlying identifier. (For example, see Section 4.3.)

This scheme is a lightweight variant of the GENI URN approach. Like GENI, it uses DNS names because they are familiar and convenient and DNS assures that they are unique. If the security of DNS is in doubt, then we can use `https` with SSL certificates to authenticate the web servers. If we do not trust the public Certifying Authorities, then we can restrict the list of eligible CAs to a single trusted root (e.g., `ch.geni.net`). SAFE uses `curl` to fetch URLs, and it is easy to configure `curl` with a list of eligible CAs.

Moreover, the result of the URL fetch with `@` is a cryptographic identifier—a token, public key hash, or self-certifying object identifier. The trust paths for validating an identifier are no different from SAFE GENI as described in this document. Importantly, the trust paths do not rely on the security of the Web server or DNS at all: a faulty server or DNS can deny access, but it cannot subvert the integrity of the protection structure.

8.5 Resources

We have not yet discussed how SAFE might be used within GENI aggregates, beyond the initial access control check for the aggregate's API operations. A reader who has read this far will not be surprised to learn that we believe SAFE is also a good formalism for resource representations exchanged via the aggregate API.

GENI uses an XML-based resource specification language (RSpec) to describe resource requests and responses (manifests) at the aggregate API. RSpec describes virtual resources as a hierarchy of objects (infrastructure elements) with attributes—nodes, links, and interfaces—together with special syntax to represent attachments of links to named interfaces outside of the tree hierarchy. In essence, the RSpec representation is equivalent to a logical description of an object graph with per-object property lists and common relationship predicates (**has-a, linked-to**).

SAFE logic sets can replace RSpec descriptions. In the current GENI standards the RSpec documents are authenticated by transmitting them over SSL rather than signing them, while SAFE logic sets are transported as signed documents (certificates). Signing has certain advantages: signed documents are nonrepudiable and it is easy to store and share them securely.

We used logical resource descriptions rather than RSpec to build the ExoGENI federation [2], a member of the NSF GENI deployment using the ORCA control framework. ORCA/ExoGENI translates to and from RSpec at its external boundary interface to the rest of GENI, but it uses logic to represent resources internally. Logical descriptions expose useful resource properties and relationships for general inference [1]. For example, they are useful in topology embedding and to enforce semantic constraints on resource requests. Declarative support for intra-slice circuit stitching was easy to add and has been available since 2010.

its controlling authority (`rootPrincipal`).

ORCA is based on SHARP [8], which introduced secure resource leasing protocols similar to those now used in GENI (the AM API 3.0, which uses per-sliver leases). SHARP is also the basis for ticketing and brokering features in ORCA/ExoGENI. SHARP tickets and leases are digitally signed contracts incorporating resource descriptions: SAFE logic sets subsume this interchange format, and much of the ticket/lease processing in SHARP can be captured declaratively.

The original SHARP/Shirako lease manager [9] used in ORCA described slivers with simple property lists. In the GENI project we extended them to transport logical resource descriptions based on a richer data model. However, the logic models used today in ORCA/ExoGENI are based on semantic web standards [1] rather than on SAFE logic.

We have developed a SAFE logic representation of tenant resources in ExoGENI—virtual Ethernet network topologies with IP networking and attached virtual or bare-metal machines. These SAFE resource representations can serve as a basis for declarative trust decisions involving controller privileges for Software-Defined Networking (SDN), inter-slice stitching, and interconnection of tenant networks and other fixed infrastructure, such as campus IP networks. However, this topic is beyond the scope of the paper.

References

[1] I. Baldine, Y. Xin, A. Mandal, C. Heerman, J. Chase, V. Marupadi, A. Yumerefendi, and D. Irwin. Autonomic cloud network orchestration: A GENI perspective. In *GLOBECOM Workshops: 2nd IEEE International Workshop on Management of Emerging Networks and Services (MENS 2010)*, December 2010.

[2] I. Baldine, Y. Xin, A. Mandal, P. Ruth, A. Yumerefendi, and J. Chase. ExoGENI: A multi-domain infrastructure-as-a-service testbed. In *TridentCom: International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, June 2012.

[3] 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.

[4] M. Berman, J. S. Chase, L. Landweber, A. Nakao, M. Ott, D. Raychaudhuri, R. Ricci, and I. Seskar. GENI: A federated testbed for innovative network experiments. *Computer Networks*, 61(0):5 – 23, 2014. Special issue on Future Internet Testbeds Part I.

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

[6] J. Chase and V. Thummala. Secure Authorization for Federated Environments (SAFE): Overview and Progress Report. Technical Report CS-2014-003, Department of Computer Science, Duke University, 2014.

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

[8] Y. Fu, J. Chase, B. Chun, S. Schwab, and A. Vahdat. SHARP: An Architecture for Secure Resource Peering. In *Proceedings of the 19th ACM Symposium on Operating System Principles*, October 2003.

[9] D. Irwin, J. S. Chase, L. Grit, A. Yumerefendi, D. Becker, and K. G. Yocum. Sharing Networked Resources with Brokered Leases. In *Proceedings of the USENIX Technical Conference*, June 2006.

[10] 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.

[11] R. L. Morgan, S. Cantor, S. Carmody, W. Hoehn, and K. Klingenstein. Federated Security: The Shibboleth Approach. *EDUCAUSE Quarterly*, 27:12–17, 2004.