Authorization and Access Control: ABAC

Ted Faber, Stephen Schwab, and John Wroclawski

1 Introduction

GENI's goal of wide-scale collaboration on infrastructure owned by independent and diverse stakeholders stresses current access control systems to the breaking point. Challenges not well addressed by current systems include, at minimum, support for distributed identity and policy management, correctness and auditability, and approachability. The Attribute Based Access Control (ABAC) system [1, 2] is an attribute-based authorization system that combines attributes using a simple reasoning system to provide authorization that (1) expresses delegation and other authorization models efficiently and scalably; (2) provides auditing information that includes both the decision and reasoning; and (3) supports multiple authentication frameworks as entry points into the attribute space. The GENI project has taken this powerful theoretical system and matured it into a form ready for practical use.

ABAC facilitates authorization decisions by providing rules under which actors in the system, called principals, prove that they have certain attributes necessary for accessing resources. Which attributes are required for a given resource is a matter of policy, to be defined and encoded with statements that are meaningful to stakeholders yet precise enough for automatic determination of authorization. ABAC represents delegation of various forms in scalable and separable ways that can be reasoned about formally. This section introduces key security concepts and challenges underlying large-scale decentralized systems, such as GENI, and illustrates the ideas behind ABAC.

T. Faber • J. Wroclawski
USC Information Sciences Institute, Los Angeles, CA, USA

S. Schwab (✉)
USC Information Sciences Institute, Arlington, VA, USA
e-mail: schwab@isi.edu

R. McGeer et al. (eds.), *The GENI Book*, DOI 10.1007/978-3-319-33769-2_10

First, we introduce *principals*, *attributes*, and *rules* of reasoning and delegation that support authorization decisions. Using ABAC, principals can represent an individual or larger organization. An attribute is a property of a principal, created by the assertion of other principals. Each principal may define their own attribute names, and issue statements assigning those attributes as needed to express security policies. Assertions are represented as a digitally signed statement, called a credential. Principals can use a range of systems to authenticate themselves, as well as using public-key cryptography to securely exchange credentials.

A principal's requests will be the subject of authorization decisions based on attributes asserted about it by other principals. Two classes of *delegation* rules, introduced in the example below and more formally described later in the chapter, enable very concise policies to assign attributes to large numbers of principles distributed across many organizations. Service providers combine attributes with rules of delegation and inference to check security policy. Enforcement decisions are ultimately made based on whether a requestor has presented the necessary attributes, assigned either directly or indirectly via delegation rules, to *authorize* a request or action.

We next illustrate, with examples drawn from GENI, how fine-grained access control may be enforced via policies. For example, an individual GENI researcher, Ted Faber, may request to use resources controlled by policies established by authorities such as the NSF, the University of Southern California (USC), or the GENI Project Office (GPO). USC (a principal) may say (assert) that Ted Faber (a principal) is a local GENI user (attribute), and enforce a policy at USC that grants access to a local GENI Rack by checking for this attribute.

Delegation may govern attributes to set policy. For example, the GPO may assert that all USC GENI users are also "*GPO prototypers*." This rule delegates authority to USC to add to the set of GPO prototypers. In this case the delegated attribute (GPO prototypers) is given to principals who also possess the delegating attribute (USC GENI user). Resource providers can, for example, grant access to slices at many GENI Racks, by enforcing a policy that checks for GPO prototypers.

Finally, a principal may delegate at one level of indirection. The GPO may assert that any NSF PI (any principal that the NSF says is a PI by signing a credential) can designate another principal as a GENI user and that user (principal) will furthermore be treated as a GPO prototyper. The NSF can affect the set of GPO prototypers by adding or removing assertions that a particular principal is a PI.

Note the flexible and fine-grained nature of decision-making afforded by using attributes, in contrast to the limited set of choices possible in less expressive identification and access control schemes. For example, to grant access to a server or resource in existing systems, each user might be given a login account, or have their individual ssh public key added to a list of authorized keys enabling login to a privileged account. In the worst case, a user may require root access. While this situation may violate best practices, early versions of GENI required users to upload their ssh private key to tools on third-party servers, granting those tools access to their GENI resources to perform tasks such as automating data collection during

an experiment. Using ABAC, resource owners set policy via the set of attributes required rather than the set of user identities granted carte blanche.

The advantage of fine-grained policies is amplified by the indirection allowed via delegation rules. Adopting this approach eliminates an inherent source of insecurity: the need to grant overly general privilege (e.g. root access via sudo) because there is no precise way to name and enable a user to perform a specific action or access a specific resource. Moreover, a central authority is not essential to track all users and their associated privileges. Rather, different principals (users, system administrators, resource owners, etc.) may make local decisions and the collective policy defined by all these stakeholders will determine privileges.

In our GENI example, the delegated attribute (GPO prototyper) is delegated to principals who possess a set of attributes (e.g. *P* GENI user for many different principals *P*). That set is defined in terms of an authorizer attribute (NSF PI). Any principal with the *authorizer* attribute can assign the *delegated* attribute by assigning their local version of the delegating attribute (*P* GENI user where *P* has the NSF PI attribute). This links the authorizer attribute to the delegating attributes, and is a potent form of ABAC delegation called a linked attribute.

The distinction between high-level policies expressed in ABAC rules and attributes, and the low-level implementation of credentials, signing formats, validation schemes and inference algorithms provides another benefit. The authentication-and-authorization (AA) logic is entirely separated from the implementation of the facility or service itself, such as a GENI aggregate manager that uses ABAC as its authorization engine. Furthermore, a record of the decisions made and the attributes and rules that factored into each decision are always available for post-mortem audit, compliance verification, or pro-active forecasting of the impact of a policy change.

Until an authorization decision needs to be made, all of the relevant credentials can be kept locally and brought together just-in-time. Principals can also pass them around so they are pre-positioned when needed, or upload them to retrieval services for ease of accessibility. For example, when the NSF designates a PI, it may send them the signed attribute credential and also forward a bulk set of certificate updates to a central organization, e.g. a GENI clearinghouse.

Reliance on signed credentials carrying attributes instead of identities also reduces the need for each facility to maintain and securely manage a separate authentication database. Rather than supporting a large number of users directly and increasing the security risks due to account break-ins through password theft, ABAC enables a much smaller and more manageable set of policy credentials to be signed, stored and shared as needed. More importantly, potentially vulnerable passwords are replaced with credentials, offering a path to improved security overall.

This chapter describes work on Attribute-Based Access Control (ABAC) that addresses the challenge of creation, management, and implementation of rich, 'audience- appropriate' authorization and access control policy management mechanisms suitable for GENI. We argue that such next-generation policy mechanisms offer a powerful tool for securing GENI as well as future national-scale cyber infrastructures such as those identified and supported by NSF and other U.S.

Government agencies in a manner that provides effective security while fostering wide-spread use and catalyzes collaboration.

The ABAC system, currently in the process of being integrated in GENI, supports authorization policy expression and enforcement mechanisms that provide:

- Formally grounded policy definition and interpretation. ABAC is based upon rigorous underlying theory and logical formalisms and semantics. Logical underpinnings are embedded deeply within the system, while users are only exposed to authorization concepts appropriate to their role and domain of expertise;
- Capability to define common vocabulary across communities and organizations. Common, well understood vocabulary may be rapidly adopted for entities, resources, and privileges within common use cases, while preserving the extensibility required to support diverse specialized policies for specialized subcommunities;
- Auditability of requests, authorizations, and policy changes. ABAC decisions result in tangible proofs of authorization derived from distributed policies, or explicit indications of what policies or insufficient privileges resulted in a request being denied;
- Library implementations suitable for incorporation into a range of GENI and future cyber infrastructures. ABAC software provides a compact library implementation and language bindings for several of the standard programming languages used throughout the Networks, Grid, Cloud and Cyber infrastructure communities.

Together, these capabilities provide a strong foundation for the implementation of strong, secure authorization and access control capabilities within large-scale, federated cyber infrastructures, while simultaneously facilitating the key objectives of flexible collaboration and local control. This chapter describes development and prototyping efforts pursued under the GENI Trial Integration Environment in DETER (TIED) project in re-implementing and delivering the ABAC authorization system [1] as a mature technology providing an expressive, practical, distributed authorization system based on formal logic. The remainder of this chapter is structured as follows. The next section elaborates on GENI's authorization requirements and how ABAC satisfies those needs, followed by a section that presents an introduction to the logical formalism that underpins ABAC and a detailed example of using ABAC to encode GENI's "speaks-for" authorization policy. This background material is followed by sections discussing system design issues related to the incorporation of a distributed authorization framework into GENI; a description of the ABAC architecture; and a brief overview of the current ABAC implementation. The chapter concludes with a brief synopsis of future directions and work.

2 GENI Authorization Requirements

GENI's primary goal is to create a distributed laboratory for large scale networking experimentation, composed as a federation of resources owned and managed by many independent organizations. Because resources are provided and managed locally, GENI spreads out the cost of administration and maintenance. Resources are supplied to researchers on-demand in accord with each resource owners' policy.

This means that an effective authorization system for GENI must be able to express resource utilization policies that are created by distributed actors, gather and apply distributed policies when considering requests, and to produce auditable information about the reasons underlying the authorization decisions to assure stakeholders that all parties are respecting their agreements.

2.1 GENI Authorization Needs

The benefits of the GENI distributed, collaborative structure can only be realized if the system as a whole can easily incorporate new researchers and new equipment and make decisions about who can use resources. Different contributors may have different requirements on who can use what resources. Though convention and agreements can minimize these differences, the GENI cyber infrastructure as a whole will be more powerful if it can potentially include interesting resources that require more restrictive access policies than those of the most generic nodes and networks present in the resource pool.

To support GENI's growth and incorporation of national and international resources, the system must be able to accept contributed resources and services, and admit users from many different institutions of diverse types with minimal a priori negotiations. These institutions will have different policies for how their resources are used and means of identifying their users.

The authorization system is responsible for finding the rights of an identified user, finding the policy that governs access to an independently managed resource and determining if that user can carry out an action on it. GENI's size and service model requires an expressiveness that local systems often do not. In particular, the resources it allocates may be used by a set of researchers with different rights. Those rights may be delegated from leaders of a project to other members and simplicity requires that these delegations be managed locally and respected globally.

In addition to delegating rights between researchers, GENI supports long-running services that operate on users' behalf. Users must be able to delegate tools the right to act on their behalf, e.g. to *speak for* them. This *speaks-for* right is a particularly interesting requirement for the GENI authorization system, and is discussed further in Sect. 3.5.

3 Attribute Based Access Control and ABAC

Speaking generically, an attribute based access control or authorization system is any system that makes decisions about authorization based on some set of explicit attributes associated with the entity seeking authorization. One widely known form of attribute based access control is role based access control, in which the decision to allow access to a resource (for use, for configuration, or for some other purpose) is based on the *role* the requestor seeking access is playing—for example, "Sheila is granted access to administer the LDAP directory because she holds the 'DIRADM' attribute", which states that she is acting in the Directory Administrator role. Attribute based access control systems typically contrast with *identity based* access control systems, in which the fundamental information exchanged between requester and requestee, and the consequent action taken by the requestee, is based on *who* the requester is, rather than the attributes the requester may hold.[1]

Somewhat confusingly, the acronym ABAC,[2] standing for Attribute Based Access Control, is also the name given to one specific, and particularly elegant, attribute based access control system by its original developers. ABAC was developed at the theoretical level in the early 2000s at Stanford and NAI Labs[3] to address distributed authorization using simple predicate logic [1]. ABAC allows service requesters and providers to attach attributes to principals in the system, define rules for deriving one attribute from others, and express those attributes and rules in a common logical framework. ABAC then employs formal logics and proofs to implement authorization decisions based on these rules, and provides interested parties with auditable evidence of the rationale behind these decisions in the form of completed proofs.

ABAC's logic is designed around the concept of principals assigning attributes to other principals, directly or through delegation rules. Attributes are scoped by the principal assigning the attribute, meaning that two attributes with the same name, but assigned by different principals, are different attributes. Delegation rules are also issued by principals and define how an attribute scoped by the issuing principal can be derived from other attributes.

Both service requesters and service providers may be ABAC principals. Services are bound to attributes (usually attributes scoped by the service provider), so that when a principal requests a service the principal providing the service does so only if the requesting principal has the appropriate attribute. Consequently, the authorization decision consists, ultimately, of proving through formal logic that the

[1] Strictly speaking, identity based systems are a subset of attribute based systems, because "identity" can be viewed as an attribute.

[2] In this chapter, use of the capitalized ABAC acronym always refers to the specific ABAC system, rather than attribute based access control systems generally.

[3] Later renamed McAfee Research. Subsequently, this research lab was acquired by SPARTA, Inc. and operated as the Security Research Division of SPARTA.

requesting principal has the attributes required to obtain the service. We discuss ABAC's authorization logics in significantly more detail below.

To simplify integrating many practical systems' notions of principals, ABAC imposes only three constraints on principal semantics. First, a principal must be able to prove its identity. Second, when two principals refer to a third by identity, they always refer to the same unique principal. Third, a principal must be able to issue assertions about attributes that are unambiguously bound to it. A public key cryptosystem such as RSA [3, 4], where the identity of a principal is its public key, meets these constraints. A principal can perform a challenge/response authentication to prove it holds the private key; a public key always refers to the same principal; and principals can issue cryptographically signed attribute assertions.[4]

Because principals are so simple and do not require significant coordination to generate, they can be created easily and without appeal to a central authority. This is a critical requirement for a system that must scale to national or global size. Many elements can be principals in such a system, and even if we only consider humans, decentralized assignment and management is key to a scalable distributed approach.

ABAC implements a fully distributed system *authorization policy*. The authorization policy of any system utilizing ABAC is distributed because a principal's direct assignment of attributes and derivation rules are managed by the issuing principal. The system's policy as a whole is the union of those distributed policy fragments (attributes and rules) but for any given decision, only the relevant rules must be consulted. Changes to a given principal's policy are only relevant to those dealing with that principal, e.g. making a resource or service request from that specific GENI aggregate manager.

Importantly, the structure of ABAC attributes—specifically, that each attribute is scoped to a principal, and that each rule assigns one attribute—allows the ABAC logic designers to ensure that a principal is assured that it can obtain all of the data relevant to an authorization decision if each principal's store of assertions can be located [2]. This insures that the correct decision can always be reached, even with the distributed management of policy.

Finally, ABAC's well-defined, logic-based framework implements a common, system-wide semantics within which authorization decisions can be clearly and unambiguously expressed and evaluated. This use of formal logic ensures that decisions are clear and transparent, and allows for both extremely simple and extremely sophisticated authorization policies to be implemented, as required by each specific use case. To increase this flexibility, ABAC defines a *family* of logics that form a hierarchy of increasingly more complex predicate logics, each reducible to datalog [5]. Because of their close relationship to role-based authorization principles, these logics are referred to as *Role-based Trust management*, or RT, logics. RT logics are discussed further below.

[4] A misbehaving principal can undermine these properties, e.g., by sharing a private key. ABAC assumes good behavior of principals.

3.1 ABAC and GENI

ABAC meets GENI's needs because it is flexible with respect to identity representation, designed to resolve distributed policy, structures its logic to make policy discovery feasible, and as mentioned provides a proof structure that supports both unambiguous decision-making and auditing. Though GENI does not currently require this capability, ABAC also allows authorization to make use of restricted information—for example security clearances or sensitive attributes—in ways that tightly limit direct and indirect disclosure.

Conceptually, ABAC's authorization logic can be applied to principals identified and authenticated by a number of different systems, independently or simultaneously. One of our contributions to GENI is providing bindings from GENI's identity system, based on X.509 certificates, to an ABAC logic system. The GENI infrastructure allows researchers to bind to X.509 identity certificates from other identity services, including Shibboleth and the InCommon [6] attribute framework.

In GENI, most of the authorization decisions can be encoded in the simplest of ABAC's RT logics, RT0. Section 4 of this chapter discusses *libabac*, a concrete implementation of the ABAC system. This software distribution supports core ABAC functionality and a robust, efficient RT0 prover, that been integrated into the GENI software base, and is in wide use today for this purpose.

3.2 ABAC Logics

ABAC presents a family of logics designed to be simple to reason about while capturing useful authorization abstractions. All the logics are based on attaching principal-scoped attributes to other principals. The logics primarily differ in the extent to which attributes can be parameterized and the rules used to delegate attributes.

Here and in subsequent sections, we give an overview of these RT authorization logics. The reader interested in a more detailed discussion of these logics is referred to Refs. 1, 2. Readers primarily focused on system implementation issues may wish to review these sections quickly before moving to Sect. 4.

ABAC defines a family of five RT logics. These include:

- **RT0**: a basic delegation logic that attaches un-parameterized attributes to principals. The basic delegation rules are direct assignment, simple delegation and linked delegation. This logic is described in detail in Sect. 3.3.
- **RT1**: RT0 extended with typed parameters attached to the attributes. Attribute parameters can be used to scope the delegation rules and further control how attributes are assigned. Described in Sect. 3.4.
- **RT2**: RT2 adds the ability to attach attributes to non-principals and reason about them. This allows one to reason about RT1 parameters using RT0 delegation rules. RT2 is described in Sect. 3.4.

- **RTT**: RT2 with the addition of a delegation rule that express consensus among some number of principals. This logic is not further discussed below.
- **RTD**: RTD with the addition of a delegation rule to delegate attributes to principals only within a specific context. This logic is not further discussed below.

Each of these logics can be expressed as datalog rules. Datalog is a negation-constrained, safe prolog subset that is efficient to implement [5].

All the logics scope their attributes by principal and share the property that a principal making a query can always ensure it can discover all delegation rules needed to reason about a request [2].

At present most of GENI's authorization needs can be met using RT0, though some forms of authorization are more elegantly and compactly represented in RT2. The GENI community has primarily focused on implementing and using these simpler logics.

The remainder of this section describes RT0, RT1, and RT2 in enough detail to give the reader a feel for their expressive power and notation. We also comment on how we use RT0 to meet GENI's needs when RT2 might be more elegant. Finally we present an extended example that shows how to use ABAC logic to express a complex GENI authorization feature, the "speaks-for" right.

3.3 RT0 Logic

ABAC's RT0 logic allows one to attach an attribute to a principal, define a direct delegation rule and define a rule linking the possession of an attribute to the ability to delegate attributes. This section introduces the notation and semantics.

In ABAC's logic an attribute is a string attached to a principal by another principal. Using a GENI example, if an aggregate manager identified as *AM* wishes to attach the *ListResources* attribute to a user identified as *U*, we say that *AM* has attached *AM.ListResources* to *U*. Only *AM* can assign attributes from the *AM* space. Furthermore *AM1.ListResources* and *AM2.ListResources* are distinct.

There are three ways to attach an attribute to a principal:

1. Direct assignment.
   Meaning: *U* has attribute *AM.ListResources*.
   Notation: $AM.ListResources \leftarrow U$
2. Delegation.
   Meaning: All principals with attribute *AM2.ListResources* have *AM1.List Resources*. Notation: $AM1.ListResources \leftarrow AM2.ListResources$
3. Linked Delegation.
   Meaning: Any principal *P* with the *AM2.Linked attribute* can assign the *AM1.ListResources* attribute by assigning the *P.ListResources* attribute.
   Notation: $AM1.ListResources \leftarrow (AM2.LinkedResources).ListResources$

Direct assignment is straightforward. A principal binds an attribute to another principal. If we take *AM.ListResources* to indicate the ability to invoke the *ListResources* operation on *AM*, the example in case 1 above is interpreted to assert that *AM* has explicitly granted that ability to user *U*.

In the second example, *AM1* has expressed a rule delegating the ability to assign principals the *AM1.ListResources* attribute to *AM2*. In turn, *AM2* exercises that delegation by assigning its *AM2.ListResources* attribute. Consequently, any principal that knows both

$AM1.ListResources \leftarrow AM2.ListResources$

and

$AM2.ListResources \leftarrow U$

can conclude that *U* has *AM1.ListResources*.

In some cases, *AM1* and *AM2* will want to closely coordinate such a delegation. In others, however, *AM2* may be entirely oblivious to the delegation. If *AM2* is a well-known certifier, or *AM1* and *AM2* have a pre-existing general relationship where they agree on the semantics of *ListResources*, there is no need or reason to discuss each specific delegation. In any case, ABAC does not require any coordination to make the delegation.

The last example above adds a second indirection. This delegates *AM1.List Resources* to a number of other principals that have an attribute assigned by *AM2*, rather than to *AM2* itself. In this case a principal must know that

$AM1.ListResources \leftarrow (AM2.Linked).ListResources$

and

$AM2.Linked \leftarrow P \text{ and } P.ListResources \leftarrow U$

to conclude that *U* has *AM1.ListResources*.

Linked delegation is best viewed as allowing a principal to appoint agents. An agent is another principal that can assign an attribute on the first principal's behalf. The example above illustrated one principal (*AM1*) directly delegating that authority to the agents of another (*AM2*). A ruleset of the form

$AM2.ListResources \leftarrow (AM2.Linked).ListResources$

$AM1.ListResources \leftarrow AM2.ListResources$

lets *AM2* express its creation of agents and *AM1* delegate to this second principal. The first rule is controlled by *AM2*, because it controls the *AM2.ListResources* attribute, while the second rule is controlled by *AM1*.

The requirements for a delegation—the right hand side of the arrows above—can include conjunctions. For example in a scenario involving a Clearing House (*CH*) and Slice Authority (*SA*),

$AM.CreateSlice \leftarrow CH.CreateSlice \cap SA.CreateSlice$

asserts that *AM* will assign the *AM.CreateSlice* attribute to a principal that has demonstrated it has both *CH.CreateSlice* and *SA.CreateSlice*. The intersection symbol $\cap$ underscores that the conjunction in the attribute interpretation is a set intersection in the set inclusion sense.

In practice, each of these declarations—the assignment of an attribute or the creation of a delegation rule, is expressed in an ABAC *credential*. The simplest

credential is a signed statement of the rule or assignment in RT0 logic, signed by the principal that controls the attribute being assigned or delegated. That is, a credential is signed by the principal whose identity is attached to the attribute on the left side of the arrow. In general, a credential may express one or several RT0 rules. Such credentials carry the assertions that form the basis of proofs in the ABAC system, and are consumable by any entity that can verify the signatures.

3.4 RT1 and RT2

RT1 adds typed parameters to attributes and the ability to reason about them. Rather than reasoning about the *AM.ListResources* attribute which might allow a principal to list any kind of resource, an RT1 rule can further scope that attribute by binding it to a named subset of resources, e.g., a particular GENI slice: *AM.ListResources(Slice1)*.

The parameters can be integers, floating-point numbers, dates, times and enumerations. The enumerations can be closed enumerations ('read', 'write', 'execute') or open-ended—for example, any principal name or file name.

ABAC can reason using parameters in three ways:

Case 1: Using literal parameters:

AM1.ListResources("Slice1") ← *AM2.ListResources("Slice1")*

Here any principal that has attribute *AM2.ListResources*, parameterized by the literal string "Slice1" also has *AM1.ListResources* parameterized by "Slice1". A principal that can list the resources of "Slice1" from principal *AM2* can also do so from principal *AM1*.

Case 2: Named parameters, implicitly constrained:

AM1.ListResources(?Slice) ← *AM2.ListResources(?Slice)*

Prefixing the parameter name with a ? marks it as a variable; the requirement for a match is that the parameter must have the same value on both sides of the assignment.[5]

For example, of a principal has attribute *AM2.ListResources("Slice1")* this rule implies that the principal also has attribute *AM1.ListResources("Slice1")*, just as in case 1. It also means that a principal with attribute *AM2.ListResources ("Some other slice")* also has attribute *AM1.ListResources("Some other slice")*.

[5]The typing is implicit. AM1.ListResources and AM2.ListResources must have direct assignments made so the system can determine the type and types must be consistent. This is a place where the theoretical nature of the ABAC papers is abundantly clear. In our implementation of RT2 we added syntax to declare types of parameters and perform explicit type checking.

To insure that the rules are tractable, any parameter name on the left hand side must also appear somewhere on the right hand side. A rule such as *AM1.ListResources(?Slice)* ← *AM2.KnownUser* is illegal.

Case 3: Named parameters, explicitly constrained (the constraint set follows the :):

AM1.ListResources(?Slice) ← *AM2.ListResources(?Slice:["Slice1", "Slice2", "Slice3"])*

This means that any principal that possesses attribute *AM2.ListResources ("Slice1")* also has attribute *AM1.ListResources("Slice1")*, and likewise for attributes parameterized by "*Slice2*" or "*Slice3*". A principal that is granted attribute *AM2.ListResources("Slice4")* is not granted any new *AM1.ListResources(?Slice)* attribute as a result.

The constraint sets must be finite, though the ABAC papers describe several syntaxes to enumerate those sets.[6]

RT1 lets policy writers naturally express authorization to certain objects for certain operations scoped by principal. It is easy to manage one principal granting another the rights to read certain objects even through complicated delegation.

RT2 relaxes the limitation that the set used to constrain parameters must be a static parts of the delegation rule. RT2 attaches principal-scoped attributes to parameter values and allows the sets of parameter values defined by those attributes to constrain parameter variables. These sets of parameter values are called *o-sets* (object sets) in the ABAC descriptions.

To see how they work, consider this rule:

AM1.ListResources(?Slice) ← *AM2.ListResources(?Slice:AM2.ValidSlice)*

This rule says that a principal that has *AM2.ListResources()* for any slice name that has the attribute *AM2.ValidSlice* also has *AM1.ListResources()* for that slice. This is basically the same rule as example 3 above, except that the set of valid slice names is dynamic.

As with principal attributes, slice name attributes are assigned by ABAC RT0 logic rules.[7] The *AM2* principal can directly declare a slice name to have *AM2.ValidSlice* by issuing the rule:

```
AM2.ValidSlice ← "Slice5"
```

The *AM2* principal may also delegate the ability to designate valid slice names to principal *AM3*:

```
AM2.ValidSlice ← AM3.ValidSlice
```

Or delegate that right to a dynamically defined set of principals:

```
AM2.ValidSlice ← (AM2.SliceNamer).ValidSlice
```

[6]For example, there is syntax for referring to the principal being evaluated when looking at a parameterized linking role. This is useful, but well beyond the scope of this document. The interested reader is referred to [1] Sections 3.1 and 3.3.

[7]In the TIED ABAC RT2 library, we use distinct notation for o-set rules and attribute rules, to ensure that each is represented by a unique type.

Moving from RT0 to RT1 requires a reasoning engine of sufficient expressive power to operate on constrained parameters. Moving from RT1 to RT2 does not require any expansion of the reasoning engine's abilities, simply the application of those features to sets of parameter values as well as sets of principals.

The rules for encoding current GENI access are generally only scoped to a given slice or sliver name. In an RT1 implementation, we would express this as *AM.DeleteSliver(uuid).* But even without an RT1 implementation, we can express this scoping within the name of an RT0 attribute, such as *AM.DeleteSliver_uuid* .

Because the rules for GENI access never require arithmetic or other operations, but only matching, and only principals who issue scoped attributes need to re-interpret them, we can express a these rules in RT0 and use a simpler reasoning engine. In the case study below, we adopt RT1 as the policy language for clarity. In practice, our implementation of "speaks-for" is based on RT0, leveraging our understanding of the GENI system to ensure soundness when these rules are, as described above, expressed in RT0 rather than RT1 format. Many GENI documents refer to this convention as RT1 Lite.

3.5 Case Study: GENI Authorization and Speaks-for

To motivate the adoption of ABAC as the primary authorization system for the GENI AM API, the authors and their team demonstrated how the existing GENI authorization model can be expressed in ABAC logic. A first step in this direction was to implement current GENI policy—including a new "speaks-for" feature—using ABAC logic [7].

This section delves into considerable detail showing how to express GENI authorization checks and "speaks-for" in RT1. The intent is to provide enough detail to convey the expressive power of the logic and illustrate the usefulness of ABAC's abstractions.

The GENI authorization model centers on specific named privileges for accessing GENI objects such as slices and resources. Resource providers grant these privileges to GENI principals (users) by issuing signed certificates that encode the privileges. These privileges are ad hoc and tied to the service definitions, and a given GENI credential could assign multiple such privileges. The specific list of privileges corresponds to operations or groups of operations supported by the GENI APIs. The GENI authorization model also allows further delegation of privileges to other principals, provided the GENI credential carrying the privilege grants that right.

The "speaks-for" privilege is an additional privilege intended to be used as follows. A user wishes to use a tool to access GENI services, but, due to security concerns, does not want to upload their identity certificate and private key to that tool, which may be a web service.

Instead, the user issues a GENI credential granting the "speaks-for" privilege to the tool (which is itself an ABAC principal). The tool includes that credential in

its requests. Consequently, the GENI services will, for authorization purposes, treat these requests from the tool as though they came directly from the user.

This differs from delegation in three ways:

1. Semantically, a tool operating under "speaks-for" authority is exercising the user's authority under close supervision. The user is taking the action through the tool and the user is responsible for the actions. In contrast, a delegated privilege is exercised independently by the recipient of that delegation. The user who has been delegated authority is responsible for its use, not the delegator.
2. All credentials may be used in conjunction with "speaks-for" authority. In contrast, credential issuers need not issue delegatable privileges.
3. A tool requires far fewer speaks-for privileges when compared to delegated privileges. (For example, a tool need not have all of a user's slice credentials to look up the status of all the user's slices.)

3.5.1 Semantics of GENI Privilege Credentials

This section describes the content and use of GENI privilege credentials [18] as used to implement speaks-for privileges using ABAC in this case study.

A GENI privilege credential encodes a set of statements of the form "The issuer of this credential (a principal) gives the owner of the credential (a principal) these privileges (strings) with respect to the target (a principal)." The privilege strings are defined with respect to the GENI APIs.

For each of the privileges, the additional optional right to delegate that privilege to others is indicated with a Boolean value. For example, a slice authority (issuer) can grant the resolve privilege to a GENI user (owner) on a given slice (target). The issuer is always the principal that signed the credential. The target and owner are given explicitly as X.509 certificates.

Under the GENI authorization semantics, a credential chain is used to encode delegation. If the credential is delegated, the original credential granting the delegatable privilege, called the base credential, is included verbatim in a new credential signed by the owner of the base credential. The owner of the base credential is the issuer of the new, delegating, credential. This new credential assigns privileges to a new owner. The new credential is valid if the base one is, if the delegated rights are marked delegatable (e.g. true), and the expiration time of the new credential does not extend beyond the expiration time of the base credential. If multiple delegations are performed, then the credential chain grows to reflect these delegations.

3.5.2 GENI Policy in RT1

Next we describe RT1 rules that express the GENI authorization policy. The policy and credential formats are entwined, and we cannot speak of one independently without the other.

Taking a different approach from the description of GENI privilege credentials above, we first describe how to encode an ABAC policy that supports "speaks-for" and subsequently extend the policy to add delegation. Speaks-for requires simpler rules to encode.

GENI Privileges with Speaks-for in RT1

For a given service request, the aggregate manager (AM) service provider knows the principal making the request, the target of the request, and which privilege is required to execute it. The service provider initializes a prover with its policy encoded as RT1 rules, augmented with any additional RT1 rules conveyed with the request. Finally, the initialized prover is queried with the question required for authorization: "does the principal making this request have the proper privilege?"

Let's examine how to encode this question in ABAC, starting with "the proper privilege." The RT1 attribute *AM.privilege(Target)* means that AM believes principals in that set have a specific privilege with respect to Target. For example, the privilege to issue *resolve* on a slice S would be the RT1 attribute *AM.resolve(S)*.

When a principal *P* requests an operation that requires resolve rights on slice *S*, the provider *AM* asks the prover if *P* is a member of *AM.resolve(S)*—or in other words, if *P* has the attribute *AM.resolve(S)*.

Service providers do not issue credentials conferring *AM.privilege()* directly to users. Instead, these privileges are inferred based on RT1 rules in the local policy. While RT1 can express complex delegation, our use case presents a series of simple delegations.

There exists a collection of issuers that each service provider trusts, and hence believes RT1 statements signed by these issuers. For each Issuer that *AM* trusts, it includes a policy rule of the form:

`AM.privilege(Target)` ← `Issuer.privilege(Target)`

There is one rule of this form for each privilege covered by AM's policy. E.g., for the privileges *resolve* and *info*, there exist corresponding rules

`AM.resolve(Target)` ← `Issuer.resolve(Target)`

`AM.info(Target)` ← `Issuer.info(Target)`

Issuers, such as Slice Authorities and Clearinghouses, issue credentials to users. Here we describe how RT1 expresses a credential as multiple RT1 statements. Because GENI policy allows all privileges to be transferred to another entity using "speaks-for", the ABAC translation of a credential issuing a privilege to a user P is expressed as:

1. *Issuer.privilege(Target)* ← *Issuer.speaks_for(P)*
2. *Issuer.speaks_for(P)* ← *P*
3. *Issuer.speaks_for(P)* ← *Issuer.TrustedTool* ∩ *P.speaks_for(P)*

These RT1 statements mean:

1. The Issuer says that anyone that speaks for P can exercise the privilege as P. Note that this means that the speaker-for, P, must also have the right under these rules. If we want only the actual principal (and whoever that principal delegates to) to have the right (1) can be modified to read *Issuer.privilege(Target)* ← *P*.
2. The Issuer says P speaks for itself.
3. The Issuer says that any entity that both the Issuer believes is a trusted tool and that P says speaks for P, speaks for P.

When an Issuer signs and delivers a GENI credential assigning privilege with respect to Target, it is making those three statements in ABAC. The first line is repeated for each privilege in the credential; the last two are added to the prover's state only once per credential.

When a user P issues a speaks-for credential for a tool T, that credential is translated into RT1 as:
`P.speaks_for(P)` ← `T`

To recap concretely in GENI terms: if T makes a request including credentials carrying all the following assertions, the corresponding rules express the policy at AM:

GENI Statement	ABAC RT1 Rules
AM trusts *Issuer* about resolve on *Target*	*AM.resolve(Target)* ← *Issuer.resolve(Target)*
Issuer has delivered to *P* a GENI privilege credential assigning resolve on *Target*	*Issuer.resolve(Target)* ← *Issuer.speaks_for(P)Issuer.speaks_for(P)* ← *PIssuer.speaks_for(P)* ← *P.speaks_for(P)*
P has issued a "speaks-for" credential to tool *T*	*P.speaks_for(P)* ← *T*
The *Issuer* trusts tool *T*	*Issuer.TrustedTool* ← *T*

When *AM* decides if *T* can proceed (e.g., if *T* is in *AM.resolve(Target)*), the proof chain supporting this inference will be:
`AM.resolve(Target)` ← `Issuer.resolve(Target)` ← `Issuer.speaks_for(P)` ← `P.speaks_for(P)` ← `T`

4 Implementing ABAC—The *libabac* System

This section describes *libabac*, an implementation of ABAC suitable for use in large, decentralized, heterogeneous distributed system designs. *Libabac* was developed by

the authors and their team for use within the DETER Cybersecurity Testbed [8] and later GENI, and is used by both of these systems today.

At time of writing, the core *libabac* distribution includes:

- Libabac itself, a linkable C/C++ library
- Perl and Python bindings to libabac
- A standalone java implementation
- *creddy*, a command line credential management tool

Two additional general-purpose tools that build on *libabac* are also available:

- *crudge* is a visual editor for ABAC policies and proofs
- *credential printer* is an XMLRPC service to convert credentials from standard system formats to a readable text representation

Libabac is both a system design architecture and a concrete implementation. At the time of writing, the concrete implementation supports a subset of the complete design architecture. This subset is sufficient to fully implement the distributed authorization policies used in both the GENI and DETER projects. However, it is expected that additional architectural elements will be incorporated into the implementation over time. Sections 4.1 and 4.2 of this chapter present system design considerations and the *libabac* system architecture, respectively, while Sect. 4.4 discusses the current implementation as incorporated into the GENI system at the time of writing.

4.1 System Design Issues

This section discusses some concrete design requirements and approaches arising from the core ABAC concept, and from issues touched on in Sects. 1 and 2. We discuss the requirements on principals, on information representation, on protocol integration and on the negotiation and collection of information.

4.1.1 Principal Requirements

In an authorization system, *principals* are the key entities on behalf of which the system carries out its function. In an attribute-based authorization system, a principal can do two things: it can ask for the system to take an action on its behalf, and it can assert attributes about itself or another principal.

For a distributed authorization system to scale, the creation of principals must be decentralized; no single entity can vet all principals or issue all identities. Consequently, what is required is a format for principal identifiers that can be issued, assigned, and managed in a decentralized fashion, and can meet three semantic requirements:

1. A principal can prove its identity to another entity.
2. If two parties reference a principal by identity, they are referencing the same principal.
3. A principal can assert attributes about other principals.

A natural implementation of these requirements is to create a public/private key pair, and make the principal's identity the public key. The principal keeps the private key secret and can prove its identity by responding to cryptographic challenges and make assertions by signing them. As long as the key space is large enough, collision probabilities can be made effectively zero, insuring a unique identity. [8]

Such a principal identifier contains minimal information in and of itself. Importantly, there is no information about the role that the principal plays, its relationship with other principals, or even a short human-readable name to print. Instead, such information is created and maintained by system applications. In particular, authorization attributes are part of the authorization framework.

Binding a Principal to a Request

Defining a principal representation sets the stage for binding principles to attributes that are used to make an authorization decision, but such a decision is also predicated on knowing which principal is requesting the action. A request must be bound to a principal and then any required attributes proven about the principal before the requestee will allow the action.

This binding is, strictly speaking, outside the responsibility of the authorization system. Formally, the system needs to know only what attribute must be proven about what principal—the binding of attributes and principals to a protocol request itself is application-specific. However, the presence of principal identifiers meeting the three requirements enumerated above requirements simplifies this task. In particular:

An interactive, channel-based request—that is, a request arriving over a specific communication channel established to support the request—can include a challenge from the requestee system to the principal that establishes the principal's identity using Property 1. Requests made across this channel can then be attributed to this principal. Network protocols based on TLS connections typically make use of this approach.

For non-interactive requests, some form of explicit binding of identity to request is required. Many practical implementations of principals—including the public/private key implementation mentioned above—allow for non-interactive binding of a principal to a request message. For example, such a message could be digitally signed and timestamped.

[8] Alternatively, a public key *fingerprint* can be utilized as the identity, if the benefits of the smaller identifier outweigh the increased collision probability.

4.1.2 Representing Attributes and Rules

Any distributed authorization system must represent the various attributes and rules that are communicated between system elements in a format that is comprehensible to each element and can be transmitted between elements effectively. Such a representation must communicate the semantics of the rule and also validate its source. In other words, when presented with a concrete representation of this logical object, a principal must be able to determine both what it means and that it is valid. We call this object a *credential*.

To support validation, the asserting principal must be bound tightly to each credential, typically through use of a digital signature. Consequently, reformatting a credential or converting between credential representations to obtain compatibility is challenging, because reformatting the credential necessarily invalidates its signature. If the conversation action is required at a point where the asserting principal is not available to resign the new version, the converted version cannot be signed correctly.

The implication for practical distributed authorization systems is that a single, or at worst a very small number, of *canonical credential formats* must be defined, which are understood by all participants, and in which credentials can be exchanged between principals and services.

While architecturally sound, this approach is unfortunately problematic from an implementation perspective, because a number of existing credential formats are already in some use today, without necessarily being either semantically sufficient or widespread enough to be adoptable as a canonical format for our purposes. At the same time, it is problematic to assume that existing credentials can be converted to some new canonical format without losing key signature information.

As examples, certain modern authentication and authorization systems, such as X.509 [9, 10] and Shibboleth/SAML [11] include the ideas of attributes and represent them naturally. In these cases, the "local" attributes render naturally into desired ABAC system attributes from a semantic perspective, but are represented in two distinct, concrete formats. Consequently, neither existing format is ideal as a canonical representation, due to the conversion issue.

Many authorization systems are based simply on identity, or alternatively do not export internal attributes used for access control, such as group membership or access control lists. For example, Kerberos acts primarily as a trusted introducer between services and clients. In a Kerberos [12] environment each service makes independent access control decisions based on trusted identity of the requesting Kerberos principal, together with internally stored authorization policy information and/or information acquired elsewhere (e.g., system group files). To integrate such information into a distributed attribute-based authorization system, we need to provide an additional translation from local information to credentials.

One way to address this discrepancy is for identity management domain to provide attribute servers. An attribute server explicitly provides credentials based on local identity. For example a Kerberized attribute service would provide, on request by a client validated by identity, a signed set of credentials in a format the

distributed authorization system understands. Similar services could be provided by enterprise systems based on password control or keyed by a PGP key. In some cases, the credential server may even generate credentials and an identity for the requesting principal. In practice, the GENI Clearinghouse is such a server.

4.1.3 Negotiation

Semantically expressive attribute-based authorization in a fully distributed environment may, in general, require a multi-party, multi-phase negotiation to complete an authorization decision. This is in contrast with the simple table lookup that characterizes local authorization. In addition, principals may need to gather credentials from others to complete the proof chains necessary to make decisions.

For an authorization system designed with this perspective, negotiation is part of the process of requesting service. As the authorization function is typically embedded within a some larger overall system, it is this larger system that must implement the protocol support for negotiated authorization function if this function is to be supported directly. Often legacy systems using pre-existing network protocols will need to be integrated as well, which demands an alternative implementation tactic. We briefly discuss those cases.

Protocol Support for Multi-Phase Negotiation

A protocol designed to support multi-phase authorization directly must include messages and function codes that indicate that a negotiation needs more information. To implement this function the application protocol must support, or be adaptable to support, a workflow as follows:

1. A service requester collects local credentials and includes them in the request.
2. The request recipient (“Server”) passes request credentials and local credentials to a reasoning engine. The server library may also attempt to gather more information from its environment to support the reasoning process.
3. If necessary, the reasoning engine returns a failure code indicating that more information is necessary and an encoding of the proof so far (as credentials).
4. The server returns to the requestor a code indicating more work is required and the proof so far.
5. The requester calls its local routines that attempt to extend the proof.
6. If the local routines have extended the proof, the requester can make the request again with the additional information.
7. When neither party can extend the proof, the request fails. Alternatively, when the proof is completed, it succeeds.

Legacy Applications and Pre-Proving

If an application utilizes existing network protocols that are not amenable to the multi-phase strategy set out above, another choice is to implement a separate pre-approval service using the same principal that offers the service. The requester asks for the same service it would request directly and interacts with the pre-approval service using a protocol designed to carry out negotiation. Once the protocol completes, either the requester and pre-approval service know that the request will not be allowed, or the requester holds the set of credentials necessary to be authorized for the action.

If the pre-approval service is conducted by the same entity providing the service, or the pre-approval service and the actual service can communicate securely, the server can be seeded with the complete proof and its result, and the requester need only provide its identity. The server binds the request to the identity and is able to "complete" the proof and authorize service for the requester.

4.1.4 Negotiating with Sensitive Data

A distributed authorization system may wish to support the notion that principals hold credentials that they may not be willing to share with all principals.

Control of these credentials is essentially a recursive application of the authorization system. A principal that holds a sensitive credential might only reveal it to a second principal if that principal can prove it has a particular required attribute.

If one principal sees that a negotiation can only be extended by revealing a sensitive credential, it inserts a sub-proof into the ongoing negotiation with the other endpoint as the target principal and the required attribute for the sensitive credential as the goal. If that sub-proof is completed, the sensitive credential can be introduced and the main proof continued.

These controls place several requirements on the negotiation. First, it introduces the requirement that all servers be principals. If controlled data is not involved, there is no reason from the negotiation semantics perspective that the server needs to be a principal.[9] The authorization decision is just a matter of collecting the proper public credentials that prove the requester has the required attribute. Because all this information is public, there is no need for the requester to restrict which entities receive its credentials. IN the case of controlled credentials, however, the requirement that all participants in a negotiation potentially be able to prove their identity mandates that all participants be principals.

Similarly, support for controlled credentials forces negotiations to be pairwise. If negotiations were even three-way, a server *Server* could construct a proof that contained information from *Client1* that *Client2* did not have permission to see, or

[9]There may be other reasons to make the server a principal; mutual authentication is rarely a mistake.

vice versa. This proof is useful to the server, but neither of the parties has permission to view it. Such a scenario undermines the transparency and logging value of the system.

It is an open problem to design a system to correctly control information in this kind of scenario. Consequently, the *libabac* system currently restricts negotiations to two parties.

4.2 libabac Software System Architecture

This section describes key aspects of the *libabac* software system architecture. It presents the basic software modules, processes, and interfaces used to implement ABAC-style authorization within *libabac*, and discusses topics related to integration of *libabac* into larger software packages. The architecture takes into account the full complexity of the ABAC authorization function, including multi-step negotiations, use of credentials gathered from third parties and negotiating with sensitive credentials.

At the time of writing this chapter, only a portion of the conceptual architecture has been implemented in the publicly available *libabac* code base. Section 4.4 provides some additional detail about the implemented *libabac* functionality.

Figure 1 shows the core components of the *libabac* implementation architecture. The application communicates with a *prover* that reasons about credentials, a *credential access controller* that controls access to sensitive credentials, and an asynchronous discovery subsystem that finds publically available credentials. These modules are linked directly with the application. Separately, the implementation provides a *credential discovery daemon*, implemented as an independent service that gathers credentials at the request of an application.

The subsystems that can be linked directly with the application (the components inside the dashed rectangle) are referred to as the *endpoint engine*. We describe key aspects of the endpoint engine's operation below.

4.2.1 Basic Operation

Applications interface with the endpoint engine primarily through the core module. The application passes relevant credentials and proof targets into the core. The core coordinates the actions of the various *libabac* subsystems to carry out as much of the proof as possible, and return either a result or a partial proof as discussed in Section "Negotiation".

From a user's perspective, the endpoint engine is primarily focused on manipulating *contexts* in which proofs are carried out. Contexts can be created, copied, deleted, and have credentials loaded into them. Each context encapsulates the knowledge directly accessible to the prover.

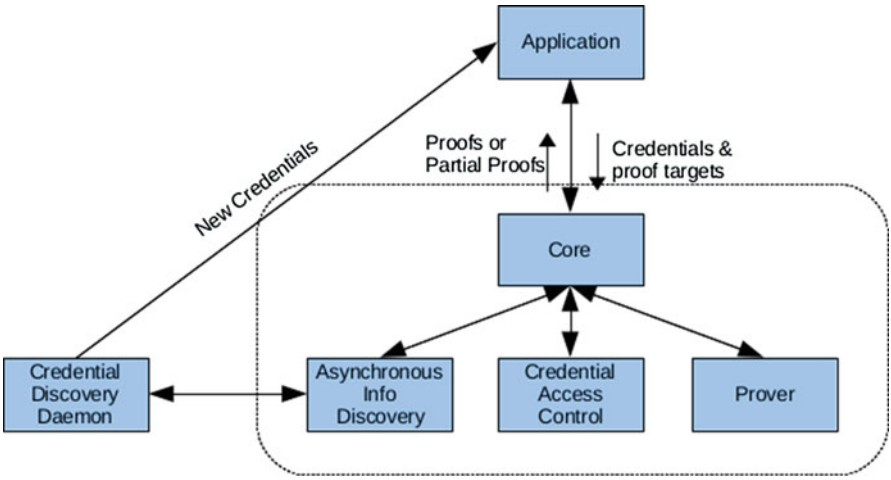

Fig. 1 *libabac* architecture components

Credentials are opaque to the user. The core provides interfaces for adding credentials to a context as required and returns them as a proof, but the user is not required to understand credential encoding. The core interface validates the credentials. An interface is provided for a user to create new credentials from an encoding-independent data structure.

When an application starts up, it creates a base context containing the credentials that define its local delegation rules as well as any other information it has stored locally or has public access to. When a request comes in, the base context is cloned into a *proving context* for the particular authorization being requested. The application binds a principal to the request as described in Section "Binding a Principal to a Request", and looks up the relevant authorization attribute for the requested action from its configuration. It then adds any credentials in the request to the proving context and makes a call into the library for the proof. The call includes the context, the target principal and the target attribute.

If the proof succeeds, the list of credentials constituting the complete proof is returned, with an indication that the proof is complete and successful. The proof is logged, the action authorized, and the proof returned to the requester for their logging purposes (assuming that the application interface allows that).

If the proof fails, a list of credentials that the prover believes are relevant to extending the proof are returned, along with an indication that the proof has failed. At this point the response is up to the application, but the architecture admits several options.

Particularly, the application may conclude that the request should not be authorized and return an appropriate error. Alternatively, the application may request an asynchronous search of public data, driven by the partial proof information. When

that search completes, the application adds the retrieved data to the proving context (or the base context) and retries the proof.

In parallel with this search, the application may return the failure message and partial proof to the requester. If the application is a pre-approver (Section "Legacy Applications and Pre-Proving") or the underlying protocol supports negotiation this would lead to a longer authorization negotiation. As each side adds information to their proof, the added information is collected in the proving context.

When neither side can extend the proof further—which an application can detect because there is no change in the size or content of the returned partial proof—the two sides must decide that the authorization has failed.

After the authorization has been concluded, the context can be deleted.

4.2.2 Asynchronous Public Data

The interface to asynchronous public data is one of the most interesting and least solidified aspects of the current *libabac* architecture. Accessing publicly available data can require long latency and may require understanding and supporting multiple externally defined query protocols. The application may want to hide this latency or place it in parallel with endpoint negotiations. The current *libabac* design encapsulates each of these alternatives.

As Fig. 1 shows, a separate process gathers credentials. It encapsulates the knowledge of outside query protocols and carries out searches asynchronously. A standard interface is defined and provided between the endpoint engine's asynchronous information discovery module and the system search daemon.

To fully meet system goals, the interface between daemon and discovery system must be generic and extensible. The discovery system must be able to guide the search, by suggesting attested attributes to search for and likely storage sites, as well as an indication of how exhaustively to search. Requests to the daemon also include an identifier so the application can route the return information to the right context. The daemon returns a list of results to the application, including the query identifier for demultiplexing them.

An important implementation consideration is the method by which these returned results are delivered to the application. In order to make *libabac* useful in as many system implementation environments as possible, it is desirable to support both traditional single threaded applications as well as more complex threaded and event-driven programming models. The *libabac* design must not limit the application designer's choices unnecessarily.

Libabac's solution is to make the interface to the asynchronous info discovery module as simple as possible. In the Unix/Linux implementation, the application makes a synchronous call into the library with hints to the search daemon, and implementations return a communications socket on which the list of returned results will be delivered. The socket itself binds requests to responses. A single-threaded application can simply block waiting for the response, while threaded and event driven programming systems should be able to easily integrate checking a

socket into their event loops and thread schedules. As the architecture matures, adapters will be included to link the response sockets into additional commonly used programming environments.

4.2.3 Controlling Sensitive Credentials

Figure 1 also shows the *Credential Access Control* block for controlling access to sensitive credentials. This block encapsulates a data vault that provides a storage facility for controlled credentials. Credentials are added to the store through the same software interface as they are added to a context, but a required attribute is also associated with each. In order to release the credential, the principal must hold the attribute.

Generally, authorization negotiations continue as previously described until the endpoint can make no further progress using its unconstrained credentials. The application can then clone the proving context and try adding each of the controlled credentials to see if the proof progresses—the prover will tell if a credential is useful, as it will be included in the output partial proof. If a controlled credential would be useful, the application will instead add the target attribute and the other principal to the list of proof targets and return that request to the other side. When the sub-goal is met—again, the prover will tell the application this—the controlled credential can be added to the mix.

In order to fully support the use of controlled credentials, the library needs to support the idea of pseudo-credentials. A pseudo-credential is a credential that the application does not have, but that would trigger a sub-proof request if the application did. The absence of a challenge could be taken as the absence of a protected credential.

Pseudo-credentials must be handled like credentials by the prover, except that a proof involving a pseudo-credential is invalid. Applications add pseudo-credentials to the controlled store, and treat them like credentials to trigger the extra proofs that hide the leakage.

4.2.4 Representing Partial Proofs

A partial proof is the representation of the current proof goals and the current state of their satisfaction. A partial proof represents the state of an authorization negotiation in progress, as described in Section “Negotiation”. The representation of a partial proof includes:

- The negotiation partners (one of which should be this entity)
- The current attribute graph (represented as credentials)
- The proof “goals”—a set of pairs indicating which attributes are to be proven about which principals
- A list of target attribute/principal pairs that will terminate the proof process

While the attribute graph shows the inferences that have been made, they are only meaningful with respect to a given goal. Any ABAC engine will connect the credentials into the same graph, but unless they agree on the proof goals, they cannot agree that it is complete, or on meaningful ways to advance the proof. There may be more than one proof goal because access control may introduce subordinate proofs, as we describe below.

The list of terminating credentials is required to indicate that some attribute proofs may be discarded if the negotiators are able to prove the real attributes of interest after discarding a subordinate objective. Only the prover cares about this list.

Although not currently part of the system design, it will likely be useful to include control interfaces that tell the prover to try to prove any one of the requested attributes or to prove all of them that it can. The prover may be able to prove a set of credentials as a set rather than serially.

4.3 Integration

The endpoint engine described above implements one side of the negotiation. An authorization decision is generally made as a result of a request for some service, and it is in that context that an endpoint engine must be embedded. Generally, we would like the endpoint engine to minimally constrain the design and operation of the larger application or service that incorporates it. Figure 2 shows two *libabac* endpoint engines working on behalf of a client and server.

The applications, client and server, primarily communicate with one another, passing requests and partial proofs between one another as described in Sect. 4.1.

If the application is using the pre-proving strategy of Section “Legacy Applications and Pre-Proving”, the legacy client and server calls out to an application set up as in Fig. 2. That application then passes the success or failure of the negotiation back to the server in a way that the legacy system understands.

4.4 A libabac Implementation

At the time of writing, a publicly available, open-source software package implementing key elements of the *libabac* conceptual system architecture has been developed by the authors and their team [13]. This implementation offers support for multiple operating systems and programming languages, and is in production use by the DETER Cybersecurity Testbed [8, 14], as well as by the GENI Program Office’s GENI toolchain. While presently providing only a subset of the full *libabac* architecture, the implementation has proven robust, flexible, and useful across a number of complex distributed authorization scenarios.

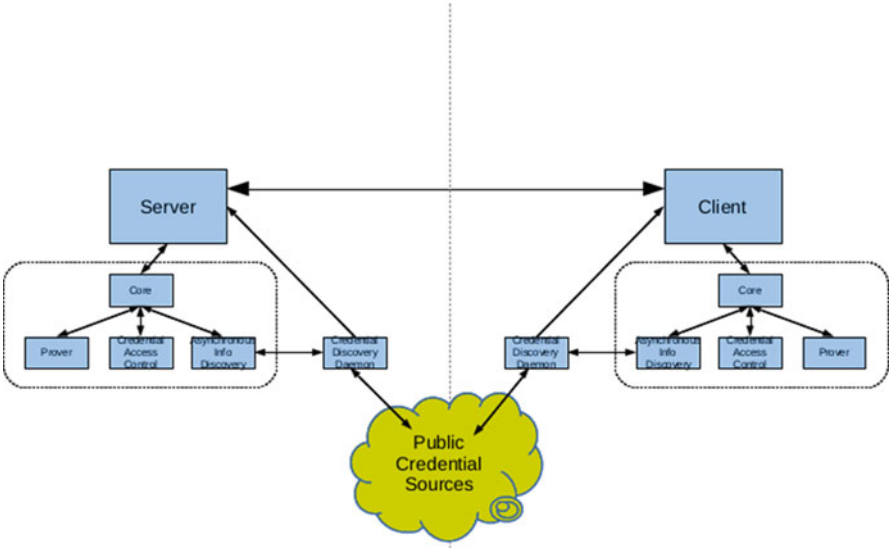

Fig. 2 Communicating applications

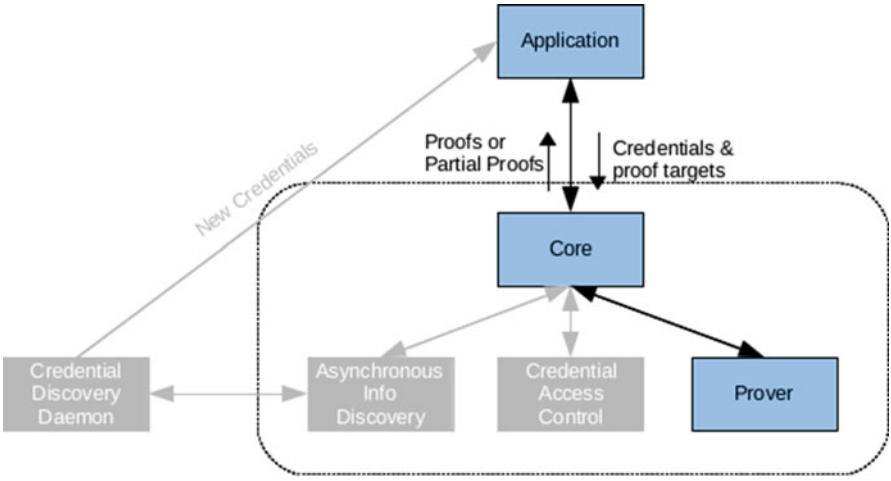

Fig. 3 *libabac* implementation components

Figure 3 shows the subset of the *libabac* conceptual system architecture available in the implementation available at time of writing. This includes an implementation of the core functions module and two independent provers—one for RT0 and one for RT2. The core and RT0 prover are well tested and in wide use within the DETER and GENI systems. The RT2 prover is presently considered experimental, with further development expected in the future.

Below, we describe some key aspects of the DETER/GENI *libabac* software package. The reader desiring further information should obtain the *libabac* software distribution [13], which includes full documentation describing the implementation and its use.

4.4.1 Core Objects

The core interface that the application uses to interact with *libabac* include implementations of the following software objects[10]:

- *Proof contexts* that hold valid credentials and can be queried to return proofs or partial proofs. Files or data buffers that encode Identities and Credentials (below) can be imported or exported from a Context. Contexts can be created, deleted and cloned. A context can be queried by asking if a principal (represented by an ID) holds a particular attribute (represented by a string). The result is success or failure along with a list of credentials that are a proof or partial proof.
- *Identities* of principals that can be used to validate credentials. If the application has appropriate information—a private key—the identity object can be used to create new credentials. These are created from a file or data buffer containing a valid X.509 certificate and optionally a matching private key.
- *Credentials* are valid attested rules (either RT0 or RT2, as appropriate) as described in Sect. 3.3. Initially, credentials were implemented using a single canonical format: as X.509 attribute certificates, an existing standard. Over time it became clear that few of the software's target users—including GENI system builders—were using this format, resulting in the conversion limitations described in Section "Representing Attributes and Rules". Consequently, the library has been expanded to accept GENI specific credential formats [19].

4.4.2 Interfaces for Creating, Managing, and Utilizing Identities and Credentials

When an authorization action is required, the typical workflow for a client is to collect its relevant credentials and identity from local files and include them with a request to a server. A server typically has a context initialized with the service's policies. When a request is passed to it, the server clones the context, imports the request's additional information, and queries the more complete context for the appropriate attribute.

The *libabac* software package provides system interfaces to support this workflow and variants. These interfaces are provided by the "core" module within the software.

[10]These are objects in the software engineering sense, containing and providing both executable methods and data.

Interfaces are also provided to create identities and credentials on the fly. Users can utilize the following interfaces to work with identities and credentials. In addition, *libabac* is packaged with simple pre-built command line and GUI tools that can perform these tasks. While intended primarily as examples, these tools are functional in their own right.

- The *Identity* object described above includes interfaces to create new identities as well. Once a new ID is created, there are interfaces to return the data needed to store the identity and its keys in files.[11]
- An *Attribute* object represents an RT0 or RT2 statement that will be turned into a credential. It features a head, representing the right hand side of the rule (the attribute being assigned) and one or more tails that represent the conditions used to assign the attribute. The head and tail are Role objects. The Attribute object can have Roles attached to it and then be baked into a Credential. Once baked, the credential data can be returned and added to a Context.
- A *Role* object represents a term in an RT0 or RT2 statement. It can be a principal (valid only on the right hand side of a type 1 rule), an attested role (valid as either a left hand side, or on the right hand side in a type 2 rule), or a linked role (valid as a right hand side of a type 3 rule). The rule types are described in Sect. 3.3.

Attributes with multiple Role objects attached to the right hand side are intersection rules.

A user who wishes to create a new identity and a credential based on it takes the following steps:

1. Create a new identity object without loading any data. The core interface will create a new ID and private key. The user may choose to save this data locally.
2. Create a new Attribute object with the ID as a basis, and the Attribute to be assigned as a Role.
3. Create Role objects for the element(s) on the right hand side and attach them to the Attribute.
4. Call the Attribute's bake interface to create a credential.
5. Get credential data from the Attribute object and either add it to a Context and carry out proofs, or save it to local storage for later use.

4.5 libabac Adoption

The *libabac* implementation discussed here is presently in use by both the GENI system and the DETER Cybersecurity Testbed. This section outlines the use of ABAC authorization and *libabac* in these systems. In both cases, ABAC concepts and the *libabac* implementation are integral to the construction, operation, and evo-

[11] In some sense this is extraneous code, as any X.509 toolkit can create an identity certificate and key files, but we have found the unified interface to be helpful.

lution of a large, heterogeneous distributed system with increasingly decentralized resource ownership and policy control.

4.5.1 Use in GENI

ABAC and *libabac* were selected by the GENI effort in 2013 to replace GENI's original ad-hoc authorization approach. This decision followed a lengthy evaluation and consideration process by the GENI Architecture Group, in which the group studied the power of the logics and state of the implementation before deciding to integrate *libabac* into the code base.

Currently GENI uses ABAC in a number of places.

- A tool to create GENI speaks-for credentials in the format libabac can process (Section "GENI Privileges with Speaks-for in RT1")
- The GENI Clearinghouse [15] evaluates these credentials when authorizing requests from tools
- The GENI Clearinghouse generates GENI credentials [17] that encode ABAC credentials directly.
- The Clearinghouse uses ABAC directly to authorize clearinghouse operations
- The GPO designers are adding ABAC authorization to Aggregate Managers, the components that directly allocate resources
- The GPO has built standalone tools for generating credentials

These systems use a mix of *libabac* software and tools written to GENI specifications that the authors and their team worked with the GPO and other stakeholders to produce. Additional tools that generate or manipulate GENI credentials, and have their own infrastructure for generating signed XML [16] have adapted that infrastructure to produce *libabac*-compatible credentials.

4.5.2 Use in DETER

The *libabac* developers are members of the DETER Cybersecurity Testbed project and have actively integrated ABAC ideas and *libabac* implementations into this facility. *Libabac* is currently fundamental to two core DETER software systems:

- The DETER Federation System (DFS) is a key element of the DETER facility. The Federation System allows DETER to create experiments that span a wide range of cyber- and cyber-physical testbed environments, each with its own use policies. DFS uses ABAC for all authorization decisions between federants. All coordination of these operations in DETER is supported directly by *libabac*.
- DETER developers are defining and implementing a new unified System Programming Interface for the DETERLab testbed and other testbed clusters that run DETER software. In this system, all testbed policies are encoded in ABAC and decisions are made using libabac.

5 Conclusions and Future Directions

GENI developers have explored the problem of distributed authorization in a national-scale distributed system and made several significant contributions to that area as part of addressing GENI's challenges. These include:

- Identifying ABAC as a viable logic to support distributed authorization.
- Definition of a software architecture to support the full expression of such a system.
- Detailed analysis of GENI's authorization needs and how they are met by both the logic and the architecture.
- A robust implementation of core ABAC functions and RT0 logic that is in current use in both GENI and DETER [8] and that will form the basis for future development in both systems.
- Prototyping an RT2 prover and studying the limitations of that prototype.

Of crucial importance, mainline system managers and operators, rather than only a small group of experts, must be able to generate policies and credentials that meet the needs of their organization and system resources. ABAC, as implemented by *libabac*, provides a firm basis for this activity, by implementing a well-defined logic that supports distributed decision making and auditing. What is needed next is a collection of tools that make this technical capability available to a much wider range of potential users.

Designing such policy and credential generating tools is the authors' key near-term future objective. These tools must be significantly more intuitive to policy designers than are existing tools that manipulate ABAC logic, which poses a challenge to policy designers and user interface designers alike. Though we have been successful in creating prototypes that capture ABAC logics directly and simple grouping and attribute assignment, we recognize that tools will need to intuitively represent more complex constructs to be useful.

Overall, GENI designers succeeded in showing that a sophisticated distributed logic can be practically applied to a national-scale system, laid out the architectural structure for future development and identified steps that will make ABAC more widely applicable.

References

1. Li, N., Mitchell, J.C., Winsborough, W.H.: Design of a role-based trust management system. In: Proceedings of the 2002 IEEE Symposium on Security and Privacy (May 2002)
2. Li, N., Winsborough, W.H., Mitchell, J.C.: Distributed credential chain discovery in trust management (extended abstract). In: Proceedings of the Eighth ACM Conference on Computer and Communications Security (CCS-8), pp. 156–165 (November 2001)
3. Callas, J., Donnerhacke, L., Finney, H., Shaw, D., Thayer, R.: Open PGP Message Format. RFC 4880 (November 2007)

4. Rivest, R., Shamir, A., Adleman, L.: A method for obtaining digital signatures and public-key cryptosystems. Commun. ACM **21**(2), 120–126 (1978)
5. Huang, S.S., Green, T.J., and Loo, B.T.: Datalog and emerging applications: an interactive tutorial. In: Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11), pp. 1213–1216. New York, NY, USA (June 2011)
6. Internet 2, InCommon: InCommon Basics and Participating in InCommon. http://www.incommon.org/docs/guides/InCommon_Resources.pdf. Retrieved Aug 2014
7. TIED Team: GENI-Compatible ABAC Credentials. http://groups.geni.net/geni/wiki/TIEDCredentials. Retrieved Aug 2014
8. ProtoGENI Team: Privileges in the Reference Implementation. http://www.protogeni.net/ProtoGeni/wiki/ReferenceImplementationPrivileges. Retrieved Aug 2014
9. Benzel, T.: The science of cyber-security experimentation: the DETER project. In: Proceedings of the Annual Computer Security Applications Conference (ACSAC) '11, Orlando, FL (December 2011)
10. Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Polk, W.: Internet X.509 Public Key Infrastructure Certificate and Certificate RevocationList (CRL) Profile. RFC 5280 (May 2008)
11. Yee, P.: Updates to the Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 6818 (January 2013)
12. Shibboleth Consortium: Shibboleth 3—A New Identity Platform. https://shibboleth.net/consortium/documents.html. Retrieved Aug 2014
13. Kohl, J., Neuman, C.: The Kerberos Network Authentication Service (V5). Internet RFC 1510 (September 1993)
14. TIED Team Libabac Software Distribution. http://abac.deterlab.net. Retrieved Aug 2014
15. The DETER Team: The DETER Federation Architecture. http://fedd.deterlab.net/wiki/FeddAbout. Retrieved Aug 2014
16. TIED Team: GENI ABAC Credentials. http://groups.geni.net/geni/wiki/TIEDABACCredential. Retrieved Aug 2014
17. GENI Program Office: Clearinghouse. http://groups.geni.net/geni/wiki/GeniClearinghouse. Retrieved Aug 2014
18. GENI Program Office: GENI Credentials. http://groups.geni.net/geni/wiki/GeniApiCredentials. Retrieved Aug 2014
19. Bartel, M., Boyer, J., Fox, B., LaMacchia, B., Simon, E.: XML Signature and Processing, 2nd edn. W3C Recommendation. http://www.w3.org/TR/xmldsig-core/ (June 2008)