Alpaca: Extensible Authorization for Distributed Services

Chris Lesniewski-Laas, Bryan Ford, Jacob Strauss, Robert Morris, and M. Frans Kaashoek
MIT CSAIL
Cambridge, MA, USA
{ctl, baford, jastr, rtm, kaashoek}@mit.edu

ABSTRACT

Traditional Public Key Infrastructures (PKI) have not lived up to their promise because there are too many ways to define PKIs, too many cryptographic primitives to build them with, and too many administrative domains with incompatible roots of trust. Alpaca is an authentication and authorization framework that embraces PKI diversity by enabling one PKI to "plug in" another PKI's credentials and cryptographic algorithms, allowing users of the latter to authenticate themselves to services using the former using their existing, unmodified certificates. Alpaca builds on Proof-Carrying Authorization (PCA) [8], expressing a credential as an explicit proof of a logical claim. Alpaca generalizes PCA to express not only delegation policies but also the cryptographic primitives, credential formats, and namespace structures needed to use foreign credentials directly. To achieve this goal, Alpaca introduces a method of creating and naming new principals which behave according to arbitrary rules, a modular approach to logical axioms, and a domain-specific language specialized for reasoning about authentication. We have implemented Alpaca as a Python module that assists applications in generating proofs (e.g., in a client requesting access to a resource), and in verifying those proofs via a compact 800-line TCB (e.g., in a server providing that resource). We present examples demonstrating Alpaca's extensibility in scenarios involving inter-organization PKI interoperability and secure remote PKI upgrade.

Categories and Subject Descriptors

D.4.6 [**Operating Systems**]: Security and Protection—*authentication, cryptographic controls*; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection—*authentication*; C.2.4 [**Computer-Communication Networks**]: Distributed Systems—*distributed applications*

General Terms

Security, Design

Keywords

security, authorization, logic, theorem-proving, cryptography

CCS'07, October 29–November 2, 2007, Alexandria, Virginia, USA.

1. INTRODUCTION

Today's deployment of public key infrastructures is surprisingly fragmented and fragile: it is not living up to PKI's potential.

The PKI vision is a rich, robust global network of trust relations, enabling anyone to present identity or access credentials to anybody else, whenever needed. This vision has been delayed by social and technological roadblocks. Organizations have adopted various authentication systems for internal use and adapted them to their own needs. The unintended result has been barriers to cross-organization compatibility, ranging from the administrative (e.g., different root certificate authorities), to the technical (e.g., different cryptosystems, credential formats, or namespaces).

The situation is not improved by the range of application-specific authentication systems. A typical IT employee might have a personal X.509 Web client certificate, a Kerberos principal, an SSH identity, a PGP key, and a plethora of passwords to online services. All of these credentials are independent, and require user attention. Moreover, if an employee of one company visits a client site, without better PKI extensibility it's unlikely that the client can even add the employee to a filesystem's access control list without manually creating an entirely new identity within the client's domain.

PKIs also need extensibility to hedge more effectively against technological change. When research reveals flaws in a cryptosystem (e.g., recent attacks on MD5 and SHA [56]), the current secure recovery path involves manual intervention on every host running the affected software. Less dramatically, pervasive software upgrades are currently required in order to adopt new kinds of cryptosystems and PKIs developed by researchers: e.g., identity-based cryptography [52], elliptic curve cryptography [38], group signatures [18], hash identifiers [37,41], multi-factor authentication [50]. Since PKI software is, by its nature, not controlled by a single entity, this upgrade process can be a serious barrier to adoption.

We believe that the world cannot be forced to adopt a single PKI, and that such a goal would be undesirable in any case. As an alternative, we propose ***Alpaca***, an extensible authorization framework that provides a "lingua franca" for authentication systems. Alpaca's lingua franca can be adopted incrementally and provides local benefit. For example, institutions can use Alpaca to provide immediate compatibility with clients and partners' PKIs without requiring all of them to adopt new certificate formats or algorithms. Although deploying Alpaca may require upgrades to client and server software, these software rollouts can usually be performed centrally, whereas the costs of incompatible certificates that Alpaca mitigates are fundamentally distributed, and thus much more difficult to control. Alpaca's lingua franca serves three goals: it accommodates the need of organizations to customize their infrastructure while minimizing their TCB, it enables them to bridge gaps between authentication systems, and it facilitates innovation in PKI technology.

Alpaca achieves these goals by building on the techniques of Proof-Carrying Authorization (PCA) [8]. The "lingua franca" is a logic language specialized with operators for expressing computation and belief [1–3, 16]. Hence, a prover presents credentials by sending a *proof* of some (application-specific) claim expressed in the logic language; the Alpaca library's credential verifier is simply a proof checker for natural deduction. The rules of deduction enable applications' authorization logic to be expressed compactly as axioms in the logic framework. While prior PCA work focused primarily on delegation policies, Alpaca generalizes the scope of PCA to include certificate and credential formats and cryptographic primitives such as hashes and signatures.

Alpaca's logic assigns a "personal namespace" to every principal using named roles: each principal controls the axioms of deduction applicable within her private "sandbox". As a result, the deployment of authentication rules need not be controlled by a central authority: anyone with an Alpaca principal name can define her own credential types by signing the appropriate verification rules. This approach doesn't risk compromising the system's integrity, since the rules' effects are confined to the principal's namespace. (Of course, she can compromise her own namespace's integrity by signing contradictory rules like "1 = 0"!)

Principals' sandboxes are inspired by the vision of a decentralized namespace in SPKI/SDSI [25, 49], in which principals are free to organize their own namespaces as they see fit. Alpaca's goal is to extend this capability beyond names, into the organization of the infrastructure itself. Consider a simple example: a service provider normally uses Alpaca RSA keys to authenticate its customers, but a prospective institutional client has already deployed identity-based cryptography (IBC) [52] keys and software to all its employees. The service provider is reluctant to expand his trusted computing base with code supporting every authentication mechanism any of his clients use. Instead, this client's administrator (or software vendor) describes the IBC signature verification algorithm in terms of Alpaca logic, and then signs that set of axioms using a regular Alpaca RSA key. The client application sends this Alpaca "meta-certificate" (a credential-describing credential) along with its requests to the provider's servers, enabling the server to authenticate these users via their IBC certificates despite the fact that the servers contain no IBC-specific code.

Alpaca proofs add little overhead to algorithms based on a few large-number arithmetic operations, such as RSA or DSA, but a high-level logic approach is inefficient when describing algorithms with many small steps, such as SHA-1, since the proof checker must verify each step. We avoid this practical issue with an efficient virtualized environment for running confined native code representing some purely computational parts of Alpaca proofs.

Alpaca's primary contribution is the generalization of PCA beyond the logic of delegation to encompass cryptographic primitives, credential decoding, and other practical components of authentication systems. Other contributions include Alpaca's unique use of named roles to construct new types of principals; a modular system for organizing axiom schemas into *authorities*; a domain-specific language specialized for reasoning about principals and constructing the proofs used in credentials; and a method of establishing an unconditionally secure root of trust from which to upgrade a system's cryptographic algorithms and keys.

Alpaca is currently implemented in 3500 lines of Python code, of which the trusted verifier is 800 lines. We demonstrate how Alpaca may be used and deployed in realistic scenarios through examples involving two fictional companies: Llamabox, a utility computing provider that needs to control and securely upgrade its global server fleet, and Spinster, a Llamabox customer that wishes to plug its customized PKI into Llamabox's system to give Spinster employees access to Llamabox servers via their usual Spinster credentials.

This paper is organized as follows. Section 2 presents the motivating example used to throughout the paper, then Section 3 details Alpaca's design and logical framework. Section 4 revisits the motivating example to demonstrate Alpaca's use in practice. Finally, Section 5 summarizes related work, Section 6 describes current implementation status and future work, and Section 7 concludes.

2. MOTIVATION

This section motivates Alpaca by exploring a hypothetical scenario that we feel is representative of many real-world situations. While we believe Alpaca is applicable to a variety of problems, for clarity we will continue referring to this example in later sections describing Alpaca's design and implementation.

2.1 Llamabox: a utility computing company

Llamabox is a company that markets distributed *utility computing* services [5–7]: on-demand access to the processing, storage, and network resources of servers that Llamabox has distributed in well-connected data centers around the world. For example, a Llamabox client that provides Web-based services to the public might respond to a temporary spike in activity in a particular city by leasing a cluster of Llamabox servers in that city and uploading replicas of their active Web content and business logic onto this on-demand hardware. Other Llamabox clients might desire longer-term access to a virtual machine running on a shared Llamabox server in each of many network locations, in order to perform large-scale network monitoring and analysis, as academic researchers currently use PlanetLab [45]. When Llamabox grants access to its computing resources to a customer, it does so by authorizing the client to create *slivers*, or private virtual resource containers analogous to PlanetLab slivers, on each of Llamabox's servers.[1]

To maximize the geographic breadth of its distributed services, Llamabox leases rack space for its servers from many different companies that run data centers around the world. Since Llamabox wishes to provide high-trust service, however, it places each of these servers in a locked equipment cabinet within the host data center, to which only Llamabox's trusted technicians have direct physical access: the host data center's regular staff only manage power, climate, and network connectivity to these locked cabinets. In the event Llamabox's central office discovers a vulnerability or compromise in any of the software on its widely-distributed servers, however—including a compromise of any of the cryptographic algorithms or keys it uses to authenticate its communication with those servers—the central office must be able to bring these compromised servers under control within seconds, reboot into a locked-down debug mode, and securely upgrade their keys and/or authentication logic. This "future-proofing" is one key capability Alpaca provides.

2.2 Spinster: an institutional client

Many of Llamabox's customers are rivals who are highly concerned about the security of Llamabox's services, and about the protection of their proprietary software and content while running on Llamabox's servers. Furthermore, these security-conscious companies already have well-established identification and authorization mechanisms that they have customized for their needs and deployed throughout their own computing infrastructure. They would

[1] A PlanetLab "sliver" is the instance of a "slice" on an individual PlanetLab server, or "node".

like their own mechanisms to extend seamlessly onto the on-demand resources they lease from Llamabox.

Spinster, a leading on-line retailer of knitting supplies, wishes to sign an institutional contract with Llamabox to secure sufficient on-demand computing resources to handle unpredictable surges in knitting enthusiasm anywhere in the world. Spinster wants to distribute its access rights to Llamabox's resources under this contract amongst Spinster's own employees, setting appropriate internal quotas on each individual's resource use to avoid budget overruns. Spinster already authenticates its employees with its own internal public-key infrastructure based on DSA keys and identity-based cryptography (IBC) keys [28, 52].

Spinster would like its own users to be able to reserve, control, and log onto Llamabox's on-demand resources using their usual Spinster certificates, just as they would use their own internal computing resources. Llamabox would like to facilitate Spinster's desire for authentication interoperability, but is not willing to risk compromising the security of its own distributed trusted computing base (TCB) by attempting to incorporate all of the cryptographic algorithms and certificate extensions that any of its customers may use. Llamabox effectively needs a secure way to allow Spinster to "plug" its custom authentication and access control logic into any of Llamabox's servers while in (shared or exclusive) use by Spinster's employees. Making independently-developed and managed authentication infrastructures interoperable in this way is the second key capability Alpaca provides.

2.3 Simplifications

A real utility computing service would have a far more complex API, authentication model, and resource model than necessary for our purposes in using Llamabox as a motivating example. To keep things simple, we consider only one resource—disk storage quota—treating Llamabox's servers essentially as shared network storage units. Correspondingly, we model slivers as fixed-size allocations of disk space on a particular server.

We simplify the security model similarly. Llamabox's central office triggers emergency reboot-and-restore in its servers via a "ping of death" packet (as in PlanetLab), whose Alpaca authentication will be discussed in Section 4. We presume that Llamabox is parsimonious in which cryptographic algorithms they "natively" support (as opposed to supporting through Alpaca's dynamic extensions). This approach reflects Llamabox's reasonable desire to minimize its TCB's dependencies.

This is the subset of interest of the Llamabox request API:

- $grant(donor, receiver, amount)$: assign a certain amount of quota from one user or entity to another. The initial source of quota is, of course, Llamabox itself.
- $create\_sliver(owner, name, size)$: create a sliver of the requested size, assigning it a sliver name and owner principal.
- $shutdown()$: the "ping of death", which brings the server down immediately, restarting it in a passive debugging mode to which only Llamabox's central office has access.

Each Llamabox server, operating independently and locally, uses Alpaca to authenticate any requests directed at it.

3. ALPACA DESIGN

This section explains how common features of authentication systems are implemented in Alpaca.

$$
\begin{array}{rcl}
\phi & := & p \ \textbf{says}\ \phi \ \mid\ \phi \rightarrow \phi \ \mid\ \forall x.\phi \ \mid\ e = e \ \mid\ sym(e, \ldots) \\
e & := & \texttt{"string"} \ \mid\ 0, 1, 2, \ldots \ \mid\ x \ \mid\ \lambda x.e \ \mid\ e(e) \\
p & := & a \ \mid\ p/e \\
a & := & \textsc{math} \ \mid\ \textsc{time} \ \mid\ \textsc{ota} \ \mid\ \ldots
\end{array}
$$

Figure 1: Grammar for Alpaca formulas (statements).

3.1 Principals

Alpaca names principals in two ways. A few atomic symbols, such as MATH, TIME, and OTA, represent principals built into Alpaca. These principals, called *authorities* (Section 3.4), don't represent users but instead follow a static set of rules.

The second kind of principal is a *named role*. Alpaca provides the role operator '/' which combines a principal with a name to get another principal: if $\mathcal{A}$ is a principal and N is any expression, then $\mathcal{A}/N$ is a new principal whose behavior is entirely controlled by $\mathcal{A}$. Roles are used in other systems (e.g., [3, 4, 8]) to enable principals to diminish their own capabilities, following the principal of least privilege [51]. Alpaca's roles have familiar semantics, but a different purpose: roles are primarily a general-purpose tool to enable any principal $\mathcal{A}$ to manufacture new types of principals $\mathcal{A}/N(\ldots)$ following rules laid out by $\mathcal{A}$. Later sections will give examples of how user principals, conjunction principals, RSA key principals, and hash principals are all implemented as roles.

3.2 Statements and proofs

Following PCA practice [8], Alpaca statements are formulas in a higher-order logic. An application request such as "client $\mathcal{A}$ wants to create a sliver named `foo` with 100 units of quota" would be represented as the formula

$$\mathcal{A}\ \textbf{says}\ \texttt{create-sliver(}\mathcal{A}\texttt{, "foo", 100)}$$

Figure 1 shows the grammar for formulas. We often use $\mathcal{A} \Rightarrow \mathcal{B}$ ("$\mathcal{A}$ speaks for $\mathcal{B}$") as shorthand for the formula

$$\forall x.\ \mathcal{A}\ \textbf{says}\ x \rightarrow\ \mathcal{B}\ \textbf{says}\ x$$

meaning that principal $\mathcal{A}$ may make statements on principal $\mathcal{B}$'s behalf. Named roles behave as if $\mathcal{A} \Rightarrow \mathcal{A}/N$ for any $\mathcal{A}$ and N.

The corresponding deductive rules (given in Appendix A) support implication, equality, lambda calculus, and universal quantification over propositions in the usual way. A *proof* is simply a valid sequence of applications of these rules leading to a true statement.

3.3 Credentials

An authentication system usually involves several parties: in the Llamabox example, this might be Llamabox's and Spinster's administrators, Spinster's employees, and Llamabox's servers. Alpaca conceptually breaks down such complex interactions into a series of two-party exchanges between an *issuer* and a *verifier*, as illustrated in Figure 2. The issuer's goal is to prove some *claim*, a statement such as $\langle \mathcal{A}\ \textbf{says}\ \texttt{create-sliver(}\mathcal{A}\texttt{,"foo",100)}\rangle$ or $\mathcal{K}_a \Rightarrow \mathcal{A}$, to the verifier. The issuing application uses the Alpaca library to generate a serialized proof of the claim, which we call a *credential*, and sends the credential over an untrusted network to the verifier in an application-specific way. The verifying application passes the credential to Alpaca's proof checker; if the credential passes this test, the verifier can act on the claim by fulfilling the issuer's request.

Typically, an authentication protocol involves multiple Alpaca exchanges, and the issuer and verifier roles change for each ex-

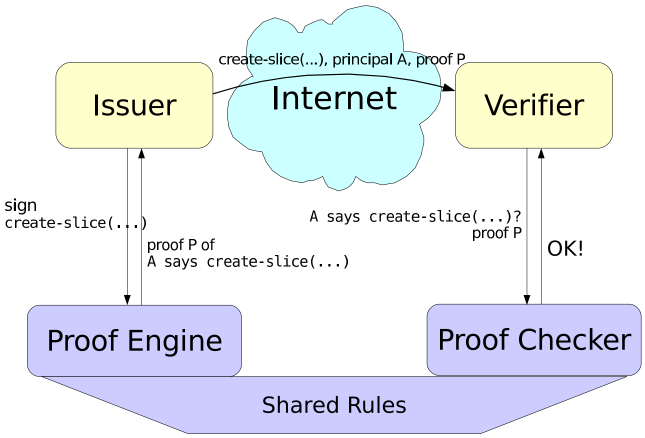

Figure 2: A client authenticates a request using Alpaca.

change. Later exchanges may build on credentials received in earlier exchanges, using them as subproofs. As an example, when Spinster first contracts with Llamabox, Llamabox may send Spinster a credential entitling Spinster to create slivers on Llamabox's servers. In this first exchange, Llamabox is the issuer and Spinster is the verifier. Then, Spinster issues Bob, an employee, a credential entitling Bob to make requests on Spinster's behalf. In this second exchange, Spinster is now the issuer and Bob is the verifier. Finally, Bob runs a client application that actually creates a sliver for Bob on one of Llamabox's servers. Bob's client uses the two credentials above as subproofs to construct a proof demonstrating that Bob is entitled to create a sliver. In the last exchange, Bob's client acts as the issuer, and Llamabox's server acts as the verifier.

3.4 Axioms

The deductive rules built in to Alpaca's logic can, by design, only prove tautologies. To prove interesting statements about principals and requests, Alpaca needs axioms, which specify how to name principals, do math, check signatures, decode "foreign" credentials, and so on. Proofs that use a particular axiom will be accepted by those Alpaca verifiers that accept that axiom.

Instead of tossing all of these axioms directly into the proof checker, Alpaca bundles them into modules called *authorities*. From a logical perspective, an authority is an abstract principal name plus a schema of axioms of the form "AUTHORITY **says** *axiom*". For example, the MATH authority consists of an infinite set of axioms of the form ⟨MATH **says** `1+2 = 3`⟩, ⟨MATH **says** `5*6 = 30`⟩, and so on. Other authorities define classes of principals by emitting axioms like

AUTHORITY **says** AUTHORITY/`princ(name)` **says** `something`

Since an authority's axioms can only affect its own "namespace", applications can choose which authorities to include in their TCB when verifying credentials: the set of trusted axioms is not fixed.

From the perspective of a credential issuer constructing a proof, an authority is a black box that takes an arbitrary list of parameters as input and produces an axiom as output. The issuer supplements the proof's chain of deductions with a list of *authority appeals*, each of which is an authority's name followed by a list of parameters. For example, the appeal (MATH, `1+2`) yields the axiom ⟨MATH **says** `1+2 = 3`⟩. Alpaca's prover engine automatically tracks dependencies on appeals when it combines credentials.

The Alpaca library implements authorities as small pluggable modules. The core library defines two generic classes, table authorities and function authorities. A *table authority* is a fixed finite list of axioms, which responds to an appeal (AUTHORITY, X) by checking that X is in the list and emitting the axiom ⟨AUTHORITY **says** X⟩. A *function authority* is a list of Python functions, which responds to an appeal of the form (AUTHORITY, $f(c_1, c_2, \dots)$) by evaluating the Python function f on the constant arguments and emitting the axiom ⟨AUTHORITY **says** $f(c_1, c_2, \dots) = c$⟩.

Table 1 lists some of the basic authorities built in to Alpaca. Some of the objects and functions defined by the function authorities such as MATH and BYTES (other than TIME) could be rebuilt within the logic system: for example, numbers can be constructed using Heyting arithmetic [55]. However, this formally minimalistic approach would add substantial practical complexity and inefficiency compared with bootstrapping from Python's arithmetic.

3.5 Establishing a secure root of trust

Alpaca's goal of making an authentication system's cryptographic algorithms extensible and replaceable presents a problem of bootstrapping: how to define, replace, or re-key a system's "master" public-key cryptographic algorithms, without relying on those same algorithms to authenticate the new rules? If Llamabox's servers normally authenticate new rules from the central office against Llamabox's master RSA key, for example, what if the central office needs to send out a new master RSA key, or switch to another master public-key algorithm?

One option is to choose a very large master RSA key and hope for the best, but we would prefer unconditionally secure authentication for a server's ultimate root of trust. Fortunately, we can build this atop unconditionally secure message authentication codes, of which a variety exist [13, 36, 54, 57]; Alpaca implements the very simplest scheme [32], similar to one-time pads for encryption.

Suppose that Llamabox embeds a separate and unique secret key, shared with its administrative center, into each server it deploys. If it becomes necessary to upgrade to a new master public key or algorithm, Llamabox sends the appropriate Alpaca rules to each server along with a "one time authenticator" (OTA) computed from that server's secret and the rules' representation. The result is that Alpaca's OTA authority produces axioms of the form ⟨OTA **says** `new-rules`⟩; section 3.7 will show how this mechanism suffices to bootstrap a new public-key algorithm.

As with one-time pads, key material for unconditionally secure MACs is "used up" as messages are sent. With the more complex schemes, this key material is used up slowly compared to the number of bytes sent. Since the logical representation of a cryptographic primitive is short, and this kind of message should be infrequent, it should not be a problem to maintain enough shared key material.

3.6 Public-key algorithms and signatures

Alpaca's built-in RSA authority identifies an RSA key having modulus n and exponent e with an Alpaca principal named RSA/`key`(n, e), which we will often abbreviate by $\mathcal{K}_{n,e}$. (We will soon see how to relax the dependency on a built-in authority.) An RSA signature in Alpaca is simply a proof of ⟨$\mathcal{K}_{n,e}$ **says** `stmt`⟩.

The single axiom we want the RSA authority to endorse is:

$$\text{RSA}\ \textbf{says}\ \forall n, e, \sigma, \phi.\, (\sigma^e = \phi \pmod{n}) \rightarrow \mathcal{K}_{n,e}\ \textbf{says}\ \phi$$

This says that if a signature σ matches a statement ϕ (using the RSA verification function), then the key's principal says that statement.

The basic approach of building the signature verification rule within the logic is one of Alpaca's unique features. Previous approaches to PCA would produce ⟨$\mathcal{K}$ **says** ϕ⟩ as an axiom when presented with a signature "out-of-band", like the special OTA au-

LOGIC	table	$\forall p, q.(p \textbf{ and } q) = (\forall r.(p \to q \to r) \to r) \qquad \forall p, q.(p \textbf{ or } q) = (\forall r.(p \to r) \to (q \to r) \to r)$
HASH	table	defines hash principals (Section 3.6.3)
MATH	function	integer operations (add, subtract, compare, multiply, modular exponent, ...)
BYTES	function	string operations (concatenate, length, substring)
PKCS1	function	octet-string-to-integer, integer-to-octet-string
TIME	function	seconds since the epoch (for credential expiration)
WITNESS	special	ensures that a concrete value has been presented (Section 3.6.2)
STATEMENT	special	relates a proposition to its string representation (Section 3.6.1)

Table 1: Important built-in authorities (axiom schemas).

thority above. Alpaca's approach enables new signature verification rules to be defined dynamically.

There are a few subtle flaws with the formulation above, however, which we'll address one by one.

3.6.1 *Representation and encoding*

In the above axiom, ϕ is an Alpaca statement, not a number, so the clause $\sigma^e = \phi \pmod n$ is nonsensical: numbers are never equal to statements. To address this issue, the special STATEMENT authority relates Alpaca statements with their string representations (serialized as canonical S-expressions). For any Alpaca statement ϕ with representation ρ, the STATEMENT authority emits

$$\text{STATEMENT} \textbf{ says } \text{STATEMENT}/\rho \textbf{ says } \phi$$

which, by $\mathcal{A} \Rightarrow \mathcal{A}/N$, is equivalent to $\langle \text{STATEMENT}/\rho \;\; \textbf{says} \;\; \phi \rangle$.[2] Since STATEMENT never produces any other sort of axiom, this means that $\text{STATEMENT}/\rho$ is essentially a principal which says only ϕ (or anything which follows from ϕ alone, e.g., $\langle X \;\; \textbf{or} \;\; \phi \rangle$).

$\langle \text{STATEMENT}/\rho \;\; \Rightarrow \mathcal{A} \rangle$ is functionally the same as $\langle \mathcal{A} \textbf{ says } \phi \rangle$. So, if we change the proposed RSA axiom from $\langle \mathcal{K}_{n,e} \textbf{ says } \phi \rangle$ to $\langle \text{STATEMENT}/\rho \;\Rightarrow\; \mathcal{K}_{n,e} \rangle$, we get the desired result.

String representations are not quite enough: the RSA verification function needs to operate on integer representations. The PKCS1 authority provides the function `octet-string-to-integer` (PKCS#1's OS2IP [35]), which converts strings to large integers.[3]

3.6.2 *Existential hazard*

What if someone can prove that, for any statement ϕ, *there exists some integer* σ that is a valid signature of ϕ? (Indeed, this is a true proposition.) This vacuous proof could be combined with the RSA axiom to prove that $\mathcal{K}_{n,e}$ says every statement ϕ, which is clearly undesirable. The WITNESS authority addresses this problem by attesting that a concrete constant has been presented by the proof:

$$\text{WITNESS} \textbf{ says } \texttt{witness}(123)$$

We can now offer a solid RSA axiom:

$$\begin{array}{l} \text{RSA} \textbf{ says } \forall n, e, \sigma, \rho. \;\; \text{WITNESS} \textbf{ says } \texttt{witness}(\sigma) \;\to \\ \quad (\sigma^e = \texttt{octet-string-to-integer}(\rho) \pmod n) \;\to \\ \quad\quad \text{STATEMENT}/\rho \;\Rightarrow\; \mathcal{K}_{n,e} \end{array}$$

Note that the built-in STATEMENT, PKCS1, and WITNESS authorities aren't specific to RSA: the functionality they provide is generic to any public-key cryptography.

3.6.3 *Hashing*

The above RSA rule can only verify statements whose representations are shorter than the RSA modulus. Alpaca's HASH authority solves this problem by defining the role $\text{HASH}/(\mathcal{A}, f, f(\rho))$, which "says" the statement encoded by some presumably collision-resistant hash function f given by the principal $\mathcal{A}$. For example, if SHA is a function authority defining the function `sha-1`, then

$$\text{HASH}/(\text{SHA}, \texttt{sha-1}, \langle \texttt{9825ccf5}\ldots \rangle)$$

is a principal that says the (nonsensical) claim $\langle \forall x.x \rangle$, because the value $\langle \texttt{9825ccf5}\ldots \rangle$ is the result of the function `sha-1` applied to the string representation of the statement $\langle \forall x.x \rangle$.

The HASH authority defines a single axiom:

$$\begin{array}{c} \forall \mathcal{A}, f, h, \rho. \;\; (\mathcal{A} \;\; \textbf{says} \;\; f(\rho) = h) \;\; \to \\ \text{STATEMENT}/\rho \;\; \Rightarrow \;\; \text{HASH}/(\mathcal{A}, f, h) \end{array}$$

A proof issuer can generate a hash principal from any statement much like she generates a signature. Afterwards, she can sign the hash by constructing and signing the compact statement

$$\forall \phi. (\text{HASH}/(\mathcal{A}, f, f(\rho)) \textbf{ says } \phi) \to \phi$$

This statement amounts to a claim that the hash principal is trusted by the principal for the RSA key used to sign the claim: anything the hash principal says, the RSA principal also says.

3.7 Dynamically creating principal types

For ease of exposition, the above sections explained how to construct key and hash principals as named roles of built-in authorities. We now show how new principal types can be introduced as named roles of other principals.

3.7.1 *A simple example: conjunction principals*

Suppose that Spinster wants the creation of new Llamabox slivers using Spinster's credentials to require the sign-off of both the CTO, Alice (with key $\mathcal{K}_a$), and the CEO, Bob (with key $\mathcal{K}_b$). Spinster's administrator (with key $\mathcal{K}_s$) can define a conjunctive principal without involving Llamabox at all, by signing this credential:

$$\begin{array}{c} \mathcal{K}_s \;\; \textbf{says} \;\; \forall \mathcal{A}, \mathcal{B}, \phi. \;\; \mathcal{A} \;\; \textbf{says} \;\; \phi \;\; \to \;\; \mathcal{B} \;\; \textbf{says} \;\; \phi \;\; \to \\ \mathcal{K}_s / \wedge (\mathcal{A}, \mathcal{B}) \;\; \textbf{says} \;\; \phi \end{array}$$

Now Spinster can delegate `create-sliver` privileges to the principal $\mathcal{K}_s / \wedge (\mathcal{K}_a, \mathcal{K}_b)$:[4]

$$\begin{array}{c} \mathcal{K}_s \;\; \textbf{says} \;\; \forall n, q. \;\; \mathcal{K}_s / \wedge (\mathcal{K}_a, \mathcal{K}_b) \textbf{ says } \texttt{create-sliver}(n, q) \;\; \to \\ \mathcal{K}_s \;\; \textbf{says} \;\; \texttt{create-sliver}(n, q) \end{array}$$

Using these two credentials together with both their keys, Alice and Bob can now construct a credential authenticating their request to create a Llamabox sliver.

[2] Free variables are not permitted in ϕ, but uninterpreted symbols are: e.g., `grant`, `create-sliver`.

[3] For brevity, this paper omits the details of PKCS#1 padding, although padding poses no problem for Alpaca.

[4] This is also an example of *restricted delegation*: $\mathcal{K}_a \wedge \mathcal{K}_b$ speaks for $\mathcal{K}_s$ only for statements of the form $\texttt{create-sliver}(n, q)$.

3.7.2 Certificate Authority

Alpaca has no built-in notion of "user names", but any Alpaca key (e.g., $\mathcal{K}_{CA}$) can set itself up as a Certificate Authority to bind names to keys by signing credentials like:

$$\mathcal{K}_{CA} \textbf{ says } \mathcal{K}_a \Rightarrow \mathcal{K}_{CA}/\texttt{user("alice")}$$

Then, armed with this credential (a certificate), Alice can use her key $\mathcal{K}_a$ to speak as the principal $\mathcal{K}_{CA}/\texttt{user}$("alice"). If $\mathcal{K}_{CA}$ is a well-known key, such as a Verisign master key or (in our example) a Llamabox or Spinster administrator key, then clients can choose to trust its name bindings.

These certificates may be extended with standard PKI features:

- Expiration and revocation are added by conditioning the credential on the current time or revocation status, as in PCA [8].
- Hierarchical namespaces like DNSSEC [23] or SPKI/SDSI [25, 49] are obtained by applying the role construction recursively.[5]

3.7.3 Bootstrapping public key algorithms

The same technique can be used to define a new public key algorithm. For example, suppose that some of Spinster's employees use DSA keys, which Llamabox's servers do not natively understand. Spinster can publish this "meta-credential":

$$\begin{array}{l}\mathcal{K}_s \textbf{ says } \forall p,q,g,y,\ \ r,s,\ \ \rho,m.\\ \quad \textsc{witness} \textbf{ says } \texttt{witness}(r,s) \rightarrow\\ \quad 0 \ < \ r,s \ < \ q \rightarrow\\ \quad m \ = \ \texttt{octet-string-to-integer(sha-1}(\rho)) \rightarrow\\ \quad \left(r = g^{ms^{-1} \bmod q} \times y^{rs^{-1} \bmod q} \bmod p \bmod q\right) \rightarrow\\ \quad\quad \textsc{statement}/\rho \Rightarrow \mathcal{K}_s/\texttt{DSA}(p,q,g,y)\end{array}$$

Despite its apparent complexity, this logic follows the same pattern as the RSA axiom above (Section 3.6.2): if the signature verification mathematics check out, then the principal $\mathcal{K}_s/\texttt{DSA}(p,q,g,y)$ emits the statement represented by ρ.

Once the above credential is published, *anyone* can create a DSA key and start signing messages as $\mathcal{K}_s/\texttt{DSA}(p,q,g,y)$: they don't need to obtain permission from $\mathcal{K}_s$. Thus, an Alpaca principal is not necessarily responsible for statements made by its named roles.

On the other hand, Alpaca does not care that $\mathcal{K}_s$ is Spinster's administrator: Bob can as easily publish a new cryptosystem's rules as Verisign can. (Of course, $\mathcal{K}_s$ is trusted by anyone using its DSA credential, so one should be selective about from whom one takes cryptographic algorithms!) The OTA authority also uses this technique to deploy the rules for a new public-key algorithm securely without itself relying on PK signatures.

3.7.4 Bootstrapping hash functions

Deploying a new hash function is similar to a new signature scheme, and is currently relevant because of the recently-revealed weaknesses in the SHA-1 and MD5 functions [27, 56]. The HASH authority defines a general-purpose family of principals indexed on the defining principal, the function name, and the hash value. Thus, all that's required is to sign the hash function's formula:[6]

$$\mathcal{K}_s \textbf{ says } \forall x.\ \texttt{DL-hash}(x) = g^x \bmod n$$

Given a credential for this statement, anyone can create HASH/($\mathcal{K}_s$,DL-hash,h) principals in the same way as HASH/(SHA,sha-1,h) principals (Section 3.6.3).

[5] Alternatively, the CA can actually delegate parts of its namespace, such as x++"@chinchilla.spinster.com", by restricted delegation to Chinchilla's sub-CA.

[6] This hash, due to Shamir [48], is collision-resistant if n is a hard-to-factor composite and g is an element of maximum order in $\mathbb{Z}_n^*$.

3.7.5 Sandboxed functions

A typical iterative hash such as SHA-1 or MD5 presents a greater challenge for Alpaca, because evaluating the hash over a long message requires Alpaca's proof checker to check every round of the hash function laboriously over every input block. To close this gap, Alpaca uses the VX32 sandboxed execution environment [29] as a safe way to run compiled code at native speed. Instead of using Alpaca's MATH authority, one would publish the credential:

$$\mathcal{K}_s \textbf{ says } \forall x.\ \texttt{hash}(x) = \texttt{evaluate-vx}(\langle\texttt{compiled code}\rangle, x)$$

When Alpaca receives an appeal for the function evaluate-vx, it executes the provided code in a sandbox with the argument x as its input, and captures the program's output. We have validated this approach with the reference implementations of the SHA-1 and Whirlpool [47] hash functions.

We could take a similar sandboxed approach to verifying signatures for public key schemes such as DSA or RSA, but since Alpaca efficiently supports modular arithmetic, there is little to be gained, and compiled native code is harder to examine and formally validate than high-level logical expressions. The evaluate-vx primitive is simply a speedup for pure computations that would take too long to evaluate as lambda-expressions.

3.8 Joining foreign PKIs via meta-certificates

Alpaca is designed to operate alongside any number of independent PKIs (Alpaca-based or not) and make the best possible use of these "foreign" credentials. This section will explain how Alpaca makes it possible to apply name bindings from one PKI to a different PKI using *meta-certificates*: credentials of the general form

$$\forall \alpha,\kappa.\ \textit{bind-foreign}(\alpha,\kappa) \rightarrow \textit{bind-local}(\alpha,\kappa)$$

where $\mathit{bind}(\alpha,\kappa)$ is a name binding from the key κ to the name α.

3.8.1 Between Alpaca PKIs

Consider the simple case of two PKIs, both based on the Alpaca PKI from Section 3.7.2; the only difference is the root CA keys ($\mathcal{K}_A$ and $\mathcal{K}_B$). If CA $\mathcal{A}$ wants to import CA $\mathcal{B}$'s namespace wholesale, she can sign a simple meta-certificate expressing this relationship:

$$\mathcal{K}_A \textbf{ says } \forall \alpha,\kappa.\ \kappa \Rightarrow \mathcal{K}_B/\texttt{user}(\alpha) \rightarrow \kappa \Rightarrow \mathcal{K}_A/\texttt{user}(\alpha)$$

If, instead, $\mathcal{B}$'s PKI has a different name structure (e.g., a tuple of name components versus a flat string, or case-sensitive versus case-insensitive names), the meta-certificate must specify a mapping between the namespaces. For example, this meta-certificate maps from $(name, domain)$ pairs to the string $name@domain$:

$$\begin{array}{c}\mathcal{K}_A \textbf{ says } \forall \alpha,\delta,\kappa.\ \kappa \Rightarrow \mathcal{K}_B/\texttt{email}(\alpha,\delta) \rightarrow\\ \kappa \Rightarrow \mathcal{K}_A/\texttt{user}(\alpha\texttt{++"@"++}\delta)\end{array}$$

3.8.2 Between Alpaca and a non-Alpaca PKI

An Alpaca-based PKI can also import credentials from a non-Alpaca-based foreign PKI, such as X.509 or SPKI certificates; this is the basis of Alpaca's ability to act as a bridge amongst authentication systems. The principle is the same as interoperating with Alpaca-based PKIs, but the meta-certificate is more complex, since it essentially specifies a logic program to decode and interpret the foreign certificate. Simplifying considerably, the meta-certificate to delegate a namespace name@x509 to an X.509 CA looks like this:

$$\begin{array}{l}\mathcal{K}_A \textbf{ says } \forall\ \mathit{cert},\ \mathit{name},\ n,\ e.\\ \quad \mathit{x509\_cn}(\ \mathit{cert}\ ) = \mathit{name} \rightarrow\\ \quad \mathit{x509\_rsa\_key}(\ \mathit{cert}\ ) = (n,e) \rightarrow\\ \quad \mathcal{K}_{X.509CA}\ \texttt{signed}\ \mathit{cert} \rightarrow\\ \quad\quad \mathcal{K}_{n,e} \Rightarrow \mathcal{K}_A/\texttt{user}(\mathit{name}\ \texttt{++ "@x509"})\end{array}$$

This credential says that for any X.509 certificate signed by the X.509 CA's key, the certificate's common name (CN) and public key should have a binding in $\mathcal{K}_A$'s namespace.

The real implementation involves several complications:

- The *x509_cn* and *x509_rsa_key* lines become expansive Alpaca logic programs that decode the DER-encoded binary representation of an X.509 certificate and extract the common name and public key fields of the certificate.

- The `signed` line uses the same logic machinery as Alpaca's key principals (Section 3.6) to check the X.509 CA's signature on the X.509 certificate. Note that the private key is not needed to construct an Alpaca proof of $\langle \mathcal{K}\ \texttt{signed}\ X\rangle$, only the public key and the signature on X.

- The meta-certificate must account for the PKCS#1 encoding of the CA's signature on the X.509 certificate.

A meta-certificate need not be issued by a CA: since Alpaca does not distinguish between CAs and other principals, anyone can publish the set of rules for decoding a foreign credential. In fact, if the set of rules is publicly agreed upon, the representation of these rules can be hashed and the resulting hash principal, a purely virtual entity not representing any person at all, can be treated as a CA!

Interfacing between Alpaca and other PKIs is similar to the X.509 case; the main difference is the details of the credential decoding function. The ultimate benefit, however, is that users need not obtain new credentials before authenticating to an Alpaca-enabled service. As long as appropriate meta-certificates are available, their existing credentials and identities can come along with them.

3.9 API

Alpaca is designed to be used within network protocols, controlled programmatically by an application rather than interactively by a user or operating autonomously. Its API for constructing and verifying credentials—claims accompanied by proofs—is thus as important to its interface as the logical framework itself.

3.9.1 Llama Lemma Language

Although an issuing application normally generates credentials dynamically (Section 3.9.2), Alpaca provides a domain-specific language to assist the programmer in creating lemmas (proof fragments) that act as "templates" for credentials. While Alpaca's Llama Lemma Language is related to general-purpose theorem-proving environments such as LF [33, 44], Coq [20], or Isabelle [42], it is specialized to the construction of credentials for Alpaca's logic. Alpaca's LLL for example supplies constructs specifically for reasoning in the context of principals and their subsidiary named roles.

To provide a flavor of the LLL,[7] Figure 3 shows an example code fragment that proves the following lemma for RSA signatures:

$$\begin{array}{l}\forall n,e,\sigma,\rho,\phi.\ \ \textsc{witness}\ \textbf{says}\ \texttt{witness}(\sigma)\ \rightarrow\\ \quad(\sigma^e\ \texttt{=}\ \texttt{octet-string-to-integer}(\rho)\pmod{n})\ \rightarrow\\ \quad\quad\textsc{statement}/\rho\ \textbf{says}\ \phi\ \rightarrow\\ \quad\quad\quad\textsc{rsa}/\texttt{key}(n,e)\ \textbf{says}\ \phi\end{array}$$

Alpaca compiles the LLL fragment of Figure 3 into a Python object representing the above lemma, encapsulating the details of the low-level proof steps. The `given` clause indicates that the nested proof will be wrapped in $\forall n, e, \sigma, \rho, \phi$. The `assuming` clause indicates the hypotheses of the nested proof, which become the premises of the lemma: e.g., $\langle \textsc{witness}\ \textbf{says}\ \texttt{witness}(\sigma)\rangle$. The result of the nested proof becomes the consequent $\langle \mathcal{K}_{n,e}\ \textbf{says}\ \phi\rangle$. The `as` construct places the nested `deduce` commands into the "frame of reference" of the principal named $\textsc{rsa}/\texttt{key}(n,e)$, implicitly prefixing everything in this scope with "$\textsc{rsa}/\texttt{key}(n,e)$ **says**". The first `deduce` clause uses a pattern matching tactic that results in an appeal to the `rsa` authority's axiom shown earlier in Section 3.6, substituting in the appropriate expressions for the variables n, e, σ, ρ and applying the hypotheses from the context to obtain the axiom's consequent $\langle \textsc{statement}/\rho \Rightarrow \mathcal{K}_{n,e}\rangle$. The remaining `deduce` clause pattern-matches the definition of "speaks-for" in the same way to obtain the final goal statement, $\langle \mathcal{K}_{n,e}\ \textbf{says}\ \phi\rangle$.

Lemmas always have the form

$$\forall x,y.\ \texttt{p}(x,y)\ \rightarrow\ \texttt{q}(x,y)\ \rightarrow\ \texttt{r}(x,y)$$

where x and y correspond to the `given` line, and `p` and `q` correspond to the clauses in the `assuming` line. A prover can take such a lemma object L, arbitrary values V and W, and previously obtained proofs P and Q of the preconditions $\texttt{p}(V,W)$ and $\texttt{q}(V,W)$, and combine them by calling the method $L.apply(V,W,P,Q)$, obtaining a proof of $\texttt{r}(V,W)$.

3.9.2 Proof construction

In order to construct a credential, a proof issuer needs a template lemma, appropriate values for the lemma's parameters, and proofs of the lemma's preconditions. The issuer normally obtains the lemma's parameters directly from the claim to be issued: for example, it applies the `rsa-says` lemma to appropriate values of n, e, σ, ρ, and ϕ to obtain the credential $\langle \mathcal{K}_{n,e}\ \textbf{says}\ \phi\rangle$, with the values of the variables substituted into this template. The parameters ρ and σ, which don't directly appear in the claim, are obtained by encoding ϕ and signing it using RSA, respectively.

The preconditions' proofs can be obtained by appealing to an authority—more precisely, by calling the *appeal* method of the issuer's local version of the authority. The resulting proof object carries the (marshalled) appeal through any calls to *apply*; eventually the appeal travels over the network as part of the credential, and is available to the verifier's proof checker when it is needed.

Alternatively, a precondition proof P might be obtained from another credential held by the prover: this is how Alpaca combines credentials. For example, the DSA meta-certificate of Section 3.7.3 would be passed to a lemma similar to Figure 3 in order to create DSA signatures. The DSA meta-certificate would be added to the `assuming` line in Figure 3, and would take the place of the RSA axiom used by the first `deduce` line.

In summary, the code for a function that constructs a credential typically follows this pattern:

1. Use the LLL to prove a general, static lemma of the form

   $$\forall x,y,\ldots\ \texttt{p}(x,y,\ldots)\rightarrow\texttt{q}(x,y,\ldots)\rightarrow\texttt{r}(x,y,\ldots)$$

   This lemma serves as a template for the credentials, in which the values of $x, y, \ldots$ vary.

2. Choose values for the variables $x, y, \ldots$ as appropriate for the desired credential.

3. Appeal to axiom authorities, or use existing credentials, to prove the preconditions for the chosen values of the variables.

4. Call the template lemma's *apply* method with the values and the preconditions' proofs, proving $\langle \texttt{r}(x,y,\ldots)\rangle$. The resulting proof is the credential.

[7] For a summary of LLL's main constructs, please see Appendix B.

```
lemma rsa-says:
  given n, e, sig, rep, stmt:
    assuming witness says witness(sig),
             math says modexp(sig, e, n) = octet-string-to-integer(rep),
             statement/rep says stmt:
      as rsa/key(n,e):                      # enter context of principal K_{n,e}
        deduce statement/rep => rsa/key(n,e) # use RSA authority's axiom
        deduce rsa/key(n,e) says stmt        # use prior lemma ∀A,B,x.A ⇒ B → A says x → B says x
```

Figure 3: Example LLL proof of the lemma `rsa-says`.

Once the issuer has constructed a credential object, he can serialize it and send it over the network to the verifier. Normally, the credential will accompany a message containing the values for the credential's parameters $(x, y, \dots)$.

3.9.3 Proof verification

Verifying credentials is simpler than constructing them. This is intentional and natural, since the structure of the credential is defined by the issuer, and opaque to the verifier. This asymmetry is what allows issuers to be upgraded without affecting the verifier code, as long as the credential's "interface"—its logical claim and set of trusted authorities—remains unchanged.

Suppose a verifier has received, over the network, a request with an Alpaca credential. To verify the credential, the receiving application follows these steps:

1. Import the receiver's trusted authorities (axiom schemas).
2. Construct a template statement for the claim it expects to accompany the type of request received.
3. Deserialize the request, substituting the request's parameters into the template to obtain a concrete claim expression.
4. Finally, invoke the `check_claim` function to check that the proof matches the expected claim; if the proof is not valid, the `check_claim` function throws an exception.

Following is a simplified version of the Python code with which Llamabox servers validate incoming `create-sliver` requests, to check that the request indeed comes from the new slice's owner.

```
def handle_create_sliver( packet ):
  auths = [rsa, math, pkcs1, witness, statement]
  template = parse( 'λ(owner, name, size).
    owner says create-sliver(owner, name, size)' )

  owner, name, size, proof = unmarshal( packet )
  claim = template.substitute( owner, name, size )

  check_claim(claim,auths,proof) # throws if failed
  create_sliver(owner,name,size) # do operation
```

While the contents of the proof are opaque to the receiver, to protect her security she must explicitly construct the claim to verify. If the message sender were instead allowed to choose the claim statement, a malicious sender could simply select a tautology, such as $\langle x = x \rangle$. Of course, the sender could easily prove a tautology even without any trusted credentials.

4. APPLYING ALPACA

We now return to the example of Llamabox and Spinster from Section 2 to illustrate how the authentication framework we have described can help solve realistic problems. First, we show how Llamabox uses Alpaca in its daily operations to manage its server pool, and to "future-proof" its system against sudden emergencies or vulnerabilities. Second, we show how Spinster takes advantage of Alpaca to extend and customize Llamabox's authorization system for compatibility with Spinster's own system. We have implemented the authentication logic described below in a model networked Llamabox application, which we use to experiment with Alpaca and service management.

4.1 Setup: secure server management

Each of Llamabox's distributed servers runs a master control program that accepts authenticated commands over the network. (It might be preferable to run this master control program, and the Alpaca proof checker it relies upon, on a trusted platform module [26] protected from the server's main operating system.)

As in the example of Section 3.9.3, the sender of a command includes an Alpaca credential, which the receiver verifies against a claim template before executing the command. To ensure that commands can't be maliciously replayed or redirected to a different server, we augment the commands from Section 2.3 with a per-message nonce and with the target server's identifier. Before accepting a command, the server checks the nonce and command arguments against a set of previously executed commands, and rejects repeated commands, as in an RPC system. Finally, the user IDs of the original API become Alpaca principals, so that servers can authenticate commands as coming from the appropriate entity. Ultimately, the credentials sent along with the commands of Section 2.3 become the following:

- donor **says** $\mathit{grant}(\mathit{donor}, \mathit{receiver}, \mathit{amount}, \mathit{nonce}, \mathit{server})$
- owner **says** $\mathit{create\_sliver}(\mathit{owner}, \mathit{name}, \mathit{size}, \mathit{nonce}, \mathit{server})$
- $\mathcal{L}$ **says** $\mathit{shutdown}(\mathit{nonce}, \mathit{server})$

Note that `grant` and `create_sliver` need only be authenticated by the quota donor in either case, and not directly by Llamabox. Since only principals with quota can execute these commands, and all quota ultimately comes from Llamabox, this creates no danger of allowing unauthorized sliver creation.

On the other hand, the "ping of death" command $\mathit{shutdown}$ can only be issued by the "Llamabox central office principal" $\mathcal{L}$, which is typically a role of the OTA authority. Servers are initially installed with a certificate $\mathcal{K}_\ell \Rightarrow \mathcal{L}$, enabling a master RSA key $\mathcal{K}_\ell$ to issue $\mathit{shutdown}$ and other administrative commands on behalf of $\mathcal{L}$. Under normal circumstances, Llamabox's central office uses $\mathcal{K}_\ell$ to manage servers; however, to hedge against the (hopefully unlikely) possibility that $\mathcal{K}_\ell$, or the RSA algorithm itself, might be broken, Llamabox also initializes each of its servers with a one-time pad to provide an alternative, inherently secure root of trust as described in Section 3.5. This random one-time pad enables Llamabox to issue secure statements directly as the principal $\mathcal{L}$ (via OTA).

4.2 Delegation and data center operations

The flexible restricted delegation capability that Alpaca inherits from PCA is useful in a server management context. For instance, consider the basic $shutdown$ command. Llamabox can always remotely take one of its servers offline, if a security compromise or other problem is detected, by issuing a $shutdown$ to the server authenticated using the master key $\mathcal{K}_\ell$.

Llamabox distributes its servers in many data centers around the world, and although Llamabox keeps those servers in locked cabinets and does not trust the ordinary staff of those data centers to submit arbitrary commands, data center staff may legitimately need to issue an emergency shutdown command to a local Llamabox server in case of power failure, overheating, and so on. Local data center operators should thus be authorized to issue $shutdown$ commands (and *only* $shutdown$ commands) to the particular Llamabox servers residing in that data center (and *only* to those servers).

To implement this delegation, Llamabox obtains the data center operator's key $\mathcal{K}_d$, and, for each $server$ in that data center, uses $\mathcal{K}_\ell$ to generate the credential:

$$\begin{aligned}\mathcal{K}_\ell\ \textbf{says}\ \forall\texttt{nonce.}\ \mathcal{K}_d\ \textbf{says}\ &\texttt{shutdown(nonce,server)} \rightarrow\\ &\mathcal{K}_\ell\ \textbf{says}\ \texttt{shutdown(nonce,server)}\end{aligned}$$

Llamabox gives this credential to the data center operator.

Because this Alpaca credential is universally quantified over the $nonce$, the data center operator may use it any number of times, choosing a new nonce each time. The credential is restricted to a particular $server$, however, permitting the operator to shut down only that server.

The data center operator holds onto this set of credentials, and if the need arises to shut down servers, it uses these credentials along with the key $\mathcal{K}_d$ to authenticate a command of the form $\mathcal{K}_\ell$ **says** $shutdown(server, nonce)$ for each affected server.

4.3 Remote re-keying and crypto upgrades

To protect its mission-critical operations, Llamabox wishes to hedge against low-probability but high-impact events, ranging from accidental loss of the master RSA key due to operational error, to a cryptographic break or weakening of RSA due to technological change. At such an event, Llamabox at least needs to re-key, and may need to upgrade to new cryptographic primitives on all of its servers. Llamabox must be able to roll out this upgrade securely without relying on the old, suspect RSA keys or algorithm.

Llamabox could of course ship each of its distributed servers to its central office for secure re-installation and then back to the server's host data center, but doing so would cause an unacceptable interruption in service. Instead, Llamabox uses Alpaca's alternative root of trust based on one-time authentication (Section 3.5) to bootstrap a new public-key algorithm with fresh keys onto its servers.

First, Llamabox revokes the master key $\mathcal{K}_\ell$ (or the RSA algorithm itself), using the technique described by Appel and Felton [8]. If simply re-keying, Llamabox generates a new master key $\mathcal{K}'_\ell$, and uses the OTA method to cause the principal $\mathcal{L}$ to issue a new certificate $\mathcal{K}'_\ell \Rightarrow \mathcal{L}$.

Suppose instead that RSA is suspect, but DSA is still considered secure. In this case, Llamabox uses the OTA method to bootstrap the rule for DSA described in Section 3.7.3. DSA key principals can now be named as $\mathcal{L}/\texttt{DSA}(p, q, g, y)$; Llamabox can thus generate a new master DSA key, and issue a certificate allowing the master DSA key to speak for $\mathcal{L}$, restoring the status quo ante.

4.4 Cross-system interfaces: foreign credentials

Suppose that Llamabox has entered into contract with Spinster to provide specific resources, e.g., 1GB of storage space on each of Llamabox's servers. As part of this arrangement, Llamabox provides Spinster's administrators with Alpaca credentials representing this allotment, as well as client software and/or a standardized interface for making requests to Llamabox's servers. Rolling out this software to Spinster's employees is not difficult via Spinster's centralized IT department, but if Llamabox were not using Alpaca, the issue of credentials would be more inconvenient: either all of Spinster's employees would need to obtain new credentials from Llamabox and associate them somehow with their Spinster credentials, or all requests from Spinster employees would need to be proxied through a gateway server verifying the employees' credentials and presenting Spinster's credentials to the Llamabox server.

Since Llamabox uses Alpaca, Spinster's administrators (or their software vendors) can use the techniques of the previous section to import Spinster employees' credentials into Llamabox's system as foreign credentials. The administrators then delegate their resource allotments to the employees' Alpaca principals as if the employees had native Alpaca credentials. Spinster still must roll out client-side software for their employees to interface with Llamabox's servers, but Spinster avoids the cost to each of its users of having to obtain and manage separate Llamabox identities.

4.5 Delegation and resource allocation

When Spinster and Llamabox sign their service contract for 1GB of storage per Llamabox server, Spinster uses Llamabox's Alpaca-based access tools to generate $\mathcal{K}_s$, an RSA key compatible with Llamabox's infrastructure. Llamabox then uses its master key $\mathcal{K}_\ell$, and a fresh nonce, to generate the Alpaca credential:

$$\mathcal{K}_\ell\,\textbf{says}\,\forall\texttt{server.}\ \texttt{grant(}\mathcal{K}_\ell\texttt{,}\mathcal{K}_s\texttt{,1GB,nonce,server)}$$

Since the claim is universally quantified over the $server$ variable, Spinster can substitute any name to get a $grant$ request credential for that server. This Alpaca credential is effectively a chit that Spinster can present at its leisure to any Llamabox server, directing that server to create a resource record for Spinster with a quota of 1GB. Since all quota originates from Llamabox, there is no question of the request failing as long as space is available. Since the nonce is fixed by the credential, Spinster cannot cheat by presenting it more than once: Llamabox's servers reject subsequent attempts.

Instead of using this resource access credential directly, suppose that Spinster wishes to delegate to each of its own users the right to use up to 100MB on any of Llamabox's servers, using Spinster's existing PKI to authenticate these users. Spinster's administrators do this by constructing a new credential embodying this policy, and distributing it to their employees along with client-side software that can compose it with the employees' individual credentials.

Suppose that the Spinster credentials composed above in Section 4.4 have the plausible form $\mathcal{K}_s/\texttt{user}(\alpha)$, where α is simply a username. Spinster can then encode the desired delegation policy via the following credential:

$$\begin{aligned}&\mathcal{K}_s\ \textbf{says}\ \forall\texttt{server,}\alpha\texttt{.}\\ &\quad\texttt{grant(}\mathcal{K}_s\texttt{,}\mathcal{K}_s\texttt{/user(}\alpha\texttt{),100MB,nonce,server)}\end{aligned}$$

Again, the nonce is fixed by the credential so that cheaters cannot use it multiple times. Different users can present it with the same nonce, however, and both $grant$ commands will go through! This is because, as specified above in Section 4.1, the Llamabox servers check the *entire command received* for repeats, not just the nonce. Thus the two requests

$$grant(\mathcal{K}_s, \mathcal{K}_s/\texttt{user}(\alpha), \texttt{100MB}, nonce, server) \quad (1)$$
$$grant(\mathcal{K}_s, \mathcal{K}_s/\texttt{user}(\beta), \texttt{100MB}, nonce, server) \quad (2)$$

are not repeats if $\alpha \neq \beta$, even if all other parameters are identical.

To use this policy credential, Spinster's client-side software also needs the master credential assigning 1GB of space per node to $\mathcal{K}_s$. Since the Llamabox server keeps track of transitive quota assignments, there is no danger that any party can exceed her quota.

All of these resource policy credentials serve, ultimately, to give each of Spinster's employees up to 100MB of *potential* quota at each Llamabox server. To make actual use of this potential quota, a user needs first to instantiate the two appropriate $grant$ commands from the credentials to siphon 100MB of actual quota into the user's account, and then make an authenticated request to create a sliver. Once the user has a sliver, he can use it to store files and objects, or (in a more general "utility computing" model) use it to run a virtual server. Because a sliver's disk usage is charged against the owner's quota, it's impossible for Spinster as a whole to exceed 1GB of sliver disk usage on any single Llamabox server.

5. RELATED WORK

Alpaca draws inspiration from the decentralized worldview of such projects as SPKI/SDSI [25, 49], SFS [37], PGP [17], and Asbestos [24]. Our compatibility focus stems in part from concern that new decentralized authentication systems, such as OpenID [43], will eschew PKI in favor of ad-hoc correlates with identity, such as DNS name or IP address. Alpaca's design is founded on proof-carrying authorization [8], and was heavily influenced by recent work in PCA and other logics of authentication and access control.

5.1 Logics of authentication

A broad range of systems have used formal logic as a basis for making authorization decisions; Abadi [1] provides a good introduction and brief survey of this area. BAN logic [16] was an early algebra of principals and statements for modeling security protocols. These ideas have been further refined by the Calculus for Access Control [3], Core Calculus of Dependency [2], and many others. The TAOS operating system [4] authenticated keys and channels using a credential algebra based on the speaks-for operator; a proof was a set of supporting sub-credentials. Recent logic-based policy languages include Binder [21], SecPAL [12], and Daisy [19].

5.2 Proof-Carrying Authorization

Appel and Felten developed "proof-carrying authentication"[8] [8] with inspiration from Necula and Li's proof-carrying code [39]. PCA admits flexible, compact, and elegant specifications of delegation policies ranging from X.509-like bindings to SDSI/SPKI decentralized names. Alpaca extends this foundation to cover credential formats and cryptographic algorithms as well as delegation policies and authorization rules, increasing the system's scope and flexibility to express complete *authentication systems*. Alpaca uses roles to construct new principal types, whereas PCA uses roles only to restrict capabilities. Alpaca also introduces tools and APIs to enable programmers to develop and extend PCA-based systems.

Bauer *et al.* [10, 11] applied Appel and Felten's Twelf-based PCA to Web client authentication, extending the client's software to search its database of credentials recursively using various strategies, assembling a proof requested by the server automatically. Howell and Kotz [34] used PCA-like proofs and an algebra similar to TAOS's [4] to enable end-to-end authorization across application boundaries. Finally, although not proof-carrying authorization *per se*, recent work on distributed construction of proofs [9] and consumable credentials via linear logics [15, 31] have enriched the design space for logic-based access control. These new techniques are parallel and complementary to Alpaca's, and we look forward to incorporating these ideas into the next version of Alpaca.

[8]Later, they re-bound the acronym to "proof-carrying authorization," to reflect the broader applicability of the idea.

5.3 Mobile code

Sandboxed mobile code is frequently proposed as a way to improve distributed systems' flexibility; for example, Alpaca uses sandboxed code to speed up pure computations that would otherwise be orders of magnitude too slow. Active Certificates [14] applies mobile code to PKIs: an active certificate is a Java applet, signed by a CA, which acts as the CA's proxy by mediating all requests from a client to a server. This approach is promising in terms of policy flexibility, but it is unclear whether it supports decentralized, flexible PKI: compiled applets don't transparently show their policies as Alpaca credentials do, limiting composability; the signature algorithm applied to applets is necessarily fixed; and the scope is apparently limited to RPC-like interactions. A possible concern is whether a malicious active certificate, acting as a mediator, can tamper with or record the interactions between a client and a server.

5.4 Public Key Infrastructures

The most widely-deployed PKIs on the public Internet today include X.509 certificates as used for the Web and email and PGP for email [17, 46]. DNSSEC [23] is a PKI for the Domain Name System in the process of being deployed as part of the IPSEC infrastructure. Unlike X.509, DNSSEC delegates the namespace according to the DNS hierarchy for scalability.

SDSI/SPKI [25, 49] is an advanced PKI with compound principals and many other features. Names in SPKI are chained as in DNSSEC, but each user populates her own root namespace with her personal contacts in addition to well-known authorities. Thus, a SPKI name might be *Dad's dentist's spouse*; these chained names are related to other systems' named roles. Appel and Felten explained how to emulate SPKI using PCA in [8]; Alpaca has the same capability (we refer the reader to [8] for details).

6. DISCUSSION

6.1 Implementation status

Alpaca is implemented as a Python 2.4 module, with 3500 nonblank lines of code, 800 lines of comments, and 1500 lines of unit tests. The verifier's TCB contains the proof checker (500 LOC) and the built-in authorities (300 LOC).

We have been evaluating Alpaca as an authentication mechanism in the Unmanaged Internet Architecture [30], an experimental adhoc distributed name system. The PKI normally uses principals named by the cryptographic hash of a public key, as well as hierarchical decentralized names like SPKI/SDSI. Alpaca provides meta-certificates (Section 3.8) enabling the PKI's namespaces to use X.509 client certificates without built-in X.509 code: e.g., a user `bob` with an X.509 certificate issued by the MIT CA appears in the PKI under the name `bob.mit`.

As described in Section 4, we are also experimenting with resource accounting credentials and secure system bootstrapping in our model Llamabox implementation.

6.2 Proof serialization overhead

The verifier's TCB does not include the protocol engine used to decode serialized proofs, since a bug would simply cause failure to verify. Since the protocol engine is untrusted, there is wide scope for optimizing the wire protocol for proofs. Proofs can be compressed by common subexpression elimination, for example, or by using more advanced techniques [40]. A more ambitious protocol might attempt to compose the proof on the fly using cached credentials, possibly saving the effort of sending the full proof.

In this vein, we implemented common subexpression elimination for Alpaca proofs, but found that in practice the standard `zlib` library does an equally good job of compressing repeated subexpressions. Since most of the size of the proof is in repeated subexpressions representing the arguments of proof steps, rather than in the "skeleton" of the proof steps themselves, eliminating this redundancy results in a compression factor of about 10.

Currently, a typical 1024-bit RSA-signed authenticator for a 100-byte message is about 1420 bytes (after `zlib` compression).[9] The optimal size, of course, is 128 bytes for an unadorned signature, so Alpaca authenticators come with significant network and memory overhead when compared to a fixed-function cryptographic format.

While the deductive portion of the proof is highly compressible, the appeals to axiom authorities (Section 3.4) are not. These appeals typically consist of poorly compressible message parameters such as entity identifiers, message body text, nonces, public key parameters, and signature values; and since this information is not related to the structure of the proof, structural proof compression techniques [40] don't apply.

In principle, most of these parameters should be computable from the application context: for example, the request parameters will contain the sender principal's name and the message body text. Also, some parameters (such as public keys) are usually repeated in every proof sent by an issuer. In the future, we plan to modify Alpaca to take advantage of this redundancy by maintaining persistent state across multiple proofs.

6.3 Stateful protocols

Alpaca's logic supports "stateless" authentication primitives such as RSA signatures and X.509 certificates, but would have trouble expressing a multi-step protocol such as Kerberos authentication [53] or TLS handshaking [22]. At present, a multi-step protocol requires the Python application code evaluating the authorization to know the protocol's steps, limiting the scope for later extending the protocol with different messages.

Looking ahead, we would like to make it possible to express cryptographic protocols, as well as primitives, entirely within the Alpaca logic. To this end, we are considering approaches based on linear logic [15, 31].

7. SUMMARY

Alpaca is an authentication framework, based on Proof-Carrying Authorization, which promotes the language of logic as a common platform for widely varying, but interoperating, public key infrastructures. Inspired by the decentralized namespaces of SPKI/SDSI, Alpaca gives each principal its own logical "sandbox" in which it can define not only its own names, but also arbitrary rules for defining new principals and credential types. Different principals' sandboxes are visible and accessible to each other, but cannot interfere with each other's operation. While encouraging innovation within the framework, Alpaca also enables users and administrators to tie together disparate PKIs through Alpaca's unique ability to decode and interpret the credentials of other systems. By allowing users to take their existing credentials with them into the common Alpaca framework, Alpaca helps combat the trend of separate, manually maintained credentials for every service a person uses.

[9] The contributions to this size include about 660 fixed bytes (increasing with greater proof complexity), 300 message bytes (multiple copies of the signed message appear in appeals), 230 key bytes, and 230 signature bytes (also appearing multiply in appeals).

Alpaca is implemented as a Python package; the verifier's TCB is limited to ~800 lines of code. We have implemented both basic PKI functions and complex meta-certificates using Alpaca, and are experimenting with it in contexts as different as decentralized networking and centralized utility computing. In the future, we hope to extend Alpaca's logic with statefulness in order to improve its efficiency and to support a wider range of protocols.

Acknowledgements

Many thanks to Jon Howell, Butler Lampson, Ronald Rivest, and the anonymous reviewers for their insightful suggestions.

8. REFERENCES

[1] M. Abadi. Logic in access control. In *IEEE Logic in Computer Science*, June 2003.

[2] M. Abadi. Access control in a core calculus of dependency. *Electronic Notes in Theoretical Computer Science*, 172:5–31, 2007.

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

[4] M. Abadi, E. Wobber, M. Burrows, and B. Lampson. Authentication in the Taos operating system. In *ACM Symposium on Operating System Principles*, pages 256–269, The Grove Park Inn and Country Club, Asheville, NC, 1993. ACM Press.

[5] Akamai. http://www.akamai.com.

[6] Amazon Elastic Compute Cloud. http://aws.amazon.com/ec2.

[7] Amazon Simple Storage Service. http://aws.amazon.com/s3.

[8] A. W. Appel and E. W. Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, Nov. 1999.

[9] L. Bauer, S. Garriss, and M. Reiter. Distributed proving in access-control systems. *IEEE Security and Privacy*, pages 81–95, 2005.

[10] L. Bauer, M. A. Schneider, and E. W. Felten. A proof-carrying authorization system. Technical Report CS-TR-638-01, Department of Computer Science, Princeton University, Apr. 2001.

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

[12] M. Y. Becker, C. Fournet, and A. D. Gordon. SecPAL: Design and semantics of a decentralized authorization language. Technical Report MSR-TR-2006-120, Microsoft Research, Sept. 2006.

[13] M. Boesgaard, T. Christensen, and E. Zenner. Badger — a fast and provably secure MAC. In *Applied Cryptography and Network Security*, 2005.

[14] N. Borisov and E. A. Brewer. Active certificates: A framework for delegation. In *NDSS*. The Internet Society, 2002.

[15] K. D. Bowers, L. Bauer, D. Garg, F. Pfenning, and M. K. Reiter. Consumable credentials in linear-logic-based access-control systems. In *Network and Distributed System Security Symposium*, pages 143–157, Feb. 2007.

[16] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. In *Proceedings of the Royal Society*, volume 426, 1989.

[17] J. Callas, L. Donnerhacke, H. Finney, and R. Thayer. OpenPGP message format. RFC 2440 (Proposed Standard), Nov. 1998.

[18] D. Chaum and E. van Heyst. Group Signatures. In *EUROCRYPT '91*, 1991.

[19] A. Cirillo, R. Jagadeesan, C. Pitcher, and J. Riely. Do As I SaY! Programmatic access control with explicit identities. In *IEEE Computer Security Foundations Symposium*, July 2007.

[20] T. Coquand. *Une Théorie des Constructions*. PhD thesis, Université Paris, Jan. 1985.

[21] J. DeTreville. Binder, a logic-based security language. In *IEEE Security and Privacy*, 2002.

[22] T. Dierks and E. Rescorla. The TLS protocol version 1.1. draft-ietf-tls-rfc2246-bis-02.txt, Network Working Group, October 2002.

[23] D. Eastlake 3rd. Domain Name System Security Extensions. RFC 2535 (Proposed Standard), Mar. 1999.

[24] P. Efstathopoulos et al. Labels and event processes in the Asbestos operating system. In *Proceedings of the 20th Symposium on Operating Systems Principles*, Brighton, UK, October 2005.

[25] C. Ellison et al. SPKI certificate theory. RFC 2693 (Experimental), Sept. 1999.

[26] P. England, B. Lampson, J. Manferdelli, M. Peinado, and B. Willman. A trusted open platform. *Computer*, 36(7):55–62, 2003.

[27] FIPS 180-1. *Secure Hash Standard*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, April 1995.

[28] FIPS 186-2. *Digital Signature Standard (DSS)*. U.S. Department of Commerce/N.I.S.T., National Technical Information Service, Springfield, VA, January 2000.

[29] B. Ford. VXA: A virtual architecture for durable compressed archives. In *4th USENIX Conference on File and Storage Technologies (FAST '05)*, San Francisco, California, December 2005.

[30] B. Ford, J. Strauss, C. Lesniewski-Laas, S. Rhea, F. Kaashoek, and R. Morris. Persistent personal names for globally connected mobile devices. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06)*, Seattle, Washington, November 2006.

[31] D. Garg, L. Bauer, K. D. Bowers, F. Pfenning, and M. K. Reiter. A linear logic of authorization and knowledge. In *Computer Security—ESORICS 2006: 11th European Symposium on Research in Computer Security*, volume 4189 of *Lecture Notes in Computer Science*, pages 297–312, Sept. 2006.

[32] E. Gilbert, F. MacWilliams, and N. Sloane. Codes which detect deception. *Bell System Technical Journal*, 53(3):405–424, 1974.

[33] R. Harper, F. Honsell, and G. Plotkin. A framework for defining logics. In *Symposium on Logic in Computer Science*, pages 194–204. IEEE Computer Society Press, June 1987.

[34] J. Howell and D. Kotz. End-to-end authorization. In *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*, pages 151–164, 2000.

[35] J. Jonsson and B. Kaliski. Public-Key Cryptography Standards (PKCS) 1. RFC 3447, 2003.

[36] U. Maurer. Information-theoretic cryptography. *Advances in Cryptology*, 99:47–64.

[37] D. Mazières, M. Kaminsky, M. F. Kaashoek, and E. Witchel. Separating key management from file system security. In *17th ACM Symposium on Operating Systems Principles (SOSP '99)*, pages 124–139, Kiawah Island, South Carolina, December 1999.

[38] V. Miller. Use of elliptic curves in cryptography. In *CRYPTO*, 1985.

[39] G. C. Necula and P. Lee. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Langauges (POPL '97)*, pages 106–119, Paris, Jan. 1997.

[40] G. C. Necula and P. Lee. Efficient representation and validation of proofs. In *IEEE Logic in Computer Science*, pages 93–104, 1998.

[41] P. Nikander and J. Laganier. An IPv6 prefix for overlay routable cryptographic hash identifiers (ORCHID). RFC 4843, 2007.

[42] T. Nipkow and L. C. Paulson. Isabelle-91. In D. Kapur, editor, *11th International Conference on Automated Deduction*, pages 673–676, Saratoga Springs, NY, 1992. Springer-Verlag LNAI 607.

[43] OpenID. http://openid.net/.

[44] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[45] PlanetLab. http://www.planet-lab.org.

[46] E. Rescorla. HTTP Over TLS. RFC 2818 (Informational), May 2000.

[47] V. Rijmen and P. S. L. M. Barreto. The WHIRLPOOL hash function, 2001.

[48] R. Rivest. Re: The Pure Crypto Project's hash function. Message to cryptography@metzdowd.com mailing list, May 2003.

[49] R. L. Rivest and B. Lampson. SDSI – a simple distributed security infrastructure. Presented at CRYPTO'96 Rumpsession, 1996.

[50] RSA SecurID token. http://www.rsa.com.

[51] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, Sept. 1975.

[52] A. Shamir. Identity-based cryptosystems and signature schemes. In *CRYPTO*, 1984.

[53] J. G. Steiner, B. C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *USENIX Winter*, pages 191–202, 1988.

[54] D. R. Stinson. Universal hashing and authentication codes. *Design, Codes, and Cryptography*, 4(4):369–380, 1994.

[55] A. S. Troelstra. Constructivism and proof theory.

[56] X. Wang, Y. Yin, and H. Yu. Finding collisions in the full SHA-1. In *CRYPTO*, 2005.

[57] M. N. Wegman and J. L. Carter. New hash functions and their use in authentication and set equality. *Journal of Computer and System Sciences*, 22:265–279, 1981.

APPENDIX

A. BUILT-IN DEDUCTIVE PROOF RULES

$$\frac{A \qquad A \to B}{B}\ implies_{elim} \qquad \frac{\frac{A}{B}}{A \to B}\ implies_{intro}$$

$$\frac{\forall x.A(x)}{A(B)}\ forall_{elim} \qquad \frac{A}{\forall x.A}\ forall_{intro}$$

$$\frac{\cdots \lambda x.A \cdots}{\cdots \lambda y.A[x \rightsquigarrow y] \cdots}\ \alpha_{sub}$$

$$\frac{\cdots (\lambda x.A)(B) \cdots}{\cdots A[x \rightsquigarrow B] \cdots}\ \beta_{reduce} \qquad \frac{\cdots A[x \rightsquigarrow B] \cdots}{\cdots (\lambda x.A)(B) \cdots}\ \beta_{abstract}$$

$$\frac{}{A = A}\ equals_{intro} \qquad \frac{A(B) \qquad B = C}{A(C)}\ equals_{elim}$$

$$\frac{A = B}{B = A}\ equals_{sym}$$

$$\frac{A \textbf{ says } A \textbf{ says } B}{A \textbf{ says } B}\ says_{elim} \qquad \frac{B}{A \textbf{ says } B}\ says_{intro}$$

$$\frac{A \textbf{ says } B \qquad A \textbf{ says } (B \to C)}{A \textbf{ says } C}\ says_{deduce}$$

$$\frac{A \textbf{ says } B}{A/C \textbf{ says } B}\ name_{intro}$$

Note: substitutions disallow free variable capture by bound scopes.

B. THE LLAMA LEMMA LANGUAGE FOR PROOF CONSTRUCTION

B.1 Commands and blocks

Alpaca's LLL compiles high-level proofs down to a tree consisting of applications of the low-level proof rules in Appendix A. The tree can then be used by the issuer's credential construction code, or serialized to an S-expression and sent to a verifier.

Alpaca processes the LLL code one line at a time, maintaining a collection (the *context*) of previously proven theorems and their proofs. A command specifies the next goal expression to prove and add to the context, and the method to prove it. In this respect, commands are a specialized kind of the *tactics* used in interactive proof assistants.

The simplest command is:

`recall` *statement*

The trivial `recall` command simply pulls an already-proven statement from the context and "proves" it by doing nothing. If the statement is not in the context, the command will fail.

Some commands may head an indented block of statements. This kind of command sets up a new context, runs the subblock within that context, and then transforms the block's output (the last theorem proved) into a new theorem. For example:

`assuming` $premise_1, premise_2, \ldots$`:` *subblock*

The `assuming` command pushes a context containing its arguments (the premises), runs the block, and pops back out of the context. If the subblock proved a theorem Q, then the `assuming` block as a whole proves the following theorem by applying the implies-introduce rule:
$\langle premise_1 \rightarrow premise_2 \rightarrow \cdots \rightarrow Q\rangle$

B.2 A trivial example

The two commands `assuming` and `recall` are enough to prove the simplest tautology, $\langle$`x → x`$\rangle$. While this tautology isn't useful as a lemma, it's enough to demonstrate how the proof construction language works.

The programmer writes an input program like this (any text after # is a comment):

```
assuming x:    # context, initial:  {}
  recall x     # context, in block: {x}
               # context, outside:  {x → x}
```

Before the first line of the program, the context is empty, since nothing has been proven yet. Entering the `assuming x` block pushes a hypothetical context in which $\langle$`x`$\rangle$ is true. Inside the subblock, the `recall x` command does not change the context, but it outputs the "theorem" $\langle$`x`$\rangle$ (which is indeed a theorem within the subblock's context). When the `assuming` block pops back out to the top-level context, it combines the premise $\langle$`x`$\rangle$ with the context-dependent proof of $\langle$`x`$\rangle$ to prove the theorem $\langle$`x → x`$\rangle$.

B.3 Other commands

`given` x`:` *subblock*

Wraps the output of its subblock in $\forall x$, using the forall-introduce rule. For example, the following fragment proves $\langle \forall x.x \rightarrow x\rangle$:

```
given x:
  assuming x:
    recall x
```

`thus` *goal*

Does nothing to the context, but checks that the last proven theorem was *goal*. This is a useful assertion for debugging nested blocks' output, as well as clarifying the code's intent by making intermediate results explicit. For example:

```
given x:
  assuming x:
    recall x
  thus x → x # output of "assuming" block
thus ∀x. x→x # output of "given" block
```

`deduce` *goal*

Check the context for a theorem of the form $\forall x, y, \ldots A \rightarrow B \rightarrow \cdots \rightarrow G$, and substitutions for the variables $x, y, \ldots$, such that G matches *goal* and $A, B, \ldots$ are also in the context. For example, this program substitutes `a` for x in $\langle \forall x.$ `p`$(x)\rightarrow$`q`$(x)\rangle$ and then applies implies-eliminate to eliminate the premise `p(a)`:

```
assuming ∀x. p(x) → q(x), p(a):
  deduce q(a)
thus (∀x. p(x) → q(x)) → p(a) → q(a)
```

`deduce` is the most commonly used command because it is used to "apply" previously proven lemmas. For example, if the context already contains proofs of the lemmas $\langle \forall x.x < x + 1\rangle$ and $\langle \forall x, y.x < y \rightarrow y > x\rangle$, then `deduce` can be used to prove $\langle \forall z.z + 1 > z\rangle$:

```
given z:
  deduce z < z+1 # apply first lemma
  deduce z+1 > z # apply second lemma
thus ∀z. z+1 > z # output of "given" block
```

`reduce` *goal*

Compute a list of β-reductions and α-substitutions that transforms the last proven theorem into *goal*. For example, this computes the path $p((\lambda y.y)(a)) \equiv_\beta p(a) \equiv_\beta (\lambda x.p(x))(a)$:

```
assuming (λx.p(x))(a):
  reduce p( (λy.y)(a) )
```

`substitute` *goal*

Check the context for a theorem $A = B$ such that substituting A for B in the last proven theorem yields *goal*. For example:

```
assuming x + y = f(z):
  assuming p( x + y ):
    substitute p( f(z) )
```

`as` $\mathcal{A}$`:` *subblock*

Simulate the context of the principal $\mathcal{A}$, and run the subblock in that context. If $\mathcal{A}$ **says** ϕ (or ϕ itself) was proven in the outer context, then ϕ is true in the subblock's context. Also, if $\mathcal{A}$ is a named role $\mathcal{B}/N$, then outer theorems of the form $\mathcal{B}$ **says** ϕ are also imported as ϕ, since $\mathcal{B}$ always speaks for $\mathcal{B}/N$. At the end of the subblock, if the result is of the form $\mathcal{A}$ **says** ϕ, output it unchanged to the outer context. Otherwise, wrap the result in $\mathcal{A}$ **says** $(\cdots)$ and output that. A simple example:

```
assuming alice says p:
  as alice/role:
    recall p
  thus alice/role says p
thus (alice says p) → (alice/role says p)
```

In general, the `as` command is a high-level interface to the low-level says-introduce, says-eliminate, says-deduce, and name-introduce rules. The above proof simply uses the name-introduce rule to convert a statement by `alice` to a statement by `alice/role`.

`lemma` *name*

Compile the subblock and attach *name* to the output theorem. The proof construction language does not use these names, but Python code uses them to look up lemmas, and names are useful for debugging.