Distributed Authorization in Vanadium

Ankur Taly(✉) and Asim Shankar

Google Inc., Mountain View, USA
ataly@google.com, ashankar@google.com

Abstract. In this tutorial, we present an authorization model for distributed systems that operate with limited internet connectivity. Reliable internet access remains a luxury for a majority of the world's population. Even for those who can afford it, a dependence on internet connectivity may lead to sub-optimal user experiences. With a focus on decentralized deployment, we present an authorization model that is suitable for scenarios where devices right next to each other (such as a sensor or a friend's phone) should be able to communicate securely in a peer-to-peer manner. The model has been deployed as part of an open-source distributed application framework called *Vanadium*. As part of this tutorial, we survey some of the key ideas and techniques used in distributed authorization, and explain how they are combined in the design of our model.

1 Introduction

Authorization is a fundamental problem in computer security that deals with whether a request to access a resource must be granted. The decision is made by a reference monitor guarding the resource. Authorization is fairly straightforward in closed systems where all resources of interest are held on a small set of devices, and reference monitors have pre-existing relationships with all authorized principals. In these systems, authorizing a request involves identifying the principal making the request, and then verifying that this identity is allowed by the resource's access control policy. The former is called *authentication*, and the latter is called *access control*.

Authorization in distributed systems is significantly more complex as the resources are spread across a network of devices under different administrative domains [22]. Moreover, not all devices and principals in the system may know each other beforehand, making even authentication complicated. For instance, consider the fairly common scenario of a user Alice trying to play a movie from her internet video service on her television (TV). It involves the TV authorizing the request from Alice to play a movie, and the video service authorizing the request from the TV to access Alice's account. The video service may recognize only Alice, and not her TV. The TV must convince the video service that it is acting on Alice's behalf.

With the advent of the Internet of Things (IoT), various physical devices that we commonly interact with in our daily lives are controllable over the network,

A. Aldini et al. (Eds.): FOSAD VIII, LNCS 9808, pp. 139–162, 2016.
DOI: 10.1007/978-3-319-43005-8_4

and are thus part of a large distributed system. These devices range from tiny embedded devices, to wearables, to large home appliances, and automobiles. The promise of IoT lies in multiple devices interacting with each other to accomplish complex tasks for the user. For instance, a home security system may interact with security cameras and locks around the house to ensure that the house is protected from intruders at all times, and all suspicious activity is logged on the user's storage server. Securely accomplishing such tasks involves making several authorization decisions. Some of the key questions that arise are *how do devices identify each other during any interaction?*, *how do users authorize devices to act on their behalf?*, *how are access control policies defined?*

Distributed authorization is a long-standing area of research, and several mechanisms have been designed for a variety of settings. However, most IoT devices still rely on rudimentary and fragile mechanisms based on passwords and unguessable IP addresses [19,26]. Indeed, over the last few months, there have been several vulnerabilities and attacks reported on various "smart" devices [1,2,4,5]. A study [19] conducted by HP on the security of several existing IoT devices reported "insufficient authorization" as one of the top security concerns. Besides this, the study found that many devices rely on a service in the cloud for authorization. We stress that proper authorization is paramount in the IoT setting as authorization breaches can impact physical security and safety. At the same time, dependence on internet connectivity can sometimes render "smart" devices unusable. Imagine the consequences of unauthorized access to an embedded heart rate monitor, or being unable to unlock the door right in front of you due to lack of internet access.

This tutorial explores the design of an authorization model for large, open distributed systems such as IoT. The primary guiding principles behind the design of the model are *decentralized deployment* and *peer-to-peer communication*. The model does not rely on any special global authorities for brokering interactions between devices that have a network path to each other. The justification for these principles is manyfold. First, a centralized model assumes all entities implicitly trust the default global authorities. This assumption fails for the enterprise and IoT settings where it is desirable to carve out isolated and fully autonomous administrative domains for devices. For instance, a user may want to be the sole authority on all devices in her home. Similarly, an enterprise may want to maintain full control of its devices with no dependence on any external authority. Second, the central nodes in the system become an attractive target for compromise to attackers. Given the frequent security breaches at well-reputed organizations, users are justifiably weary of having third-party services store more personal data than strictly necessary. Besides this, protecting personal user data from external and internal threats is quite burdensome for the organizations as well. Finally, a centralized model requires that all devices maintain connectivity to external global services, which is infeasible for many IoT devices, including ones that communicate only over Bluetooth, or ones present in public spaces such as shopping malls, buses, and trains, where internet access is unreliable. Moreover, reliable internet access remains a luxury for a majority of the world's population, where routing most interactions through a cloud service can be expensive or simply not possible.

The authorization model presented in this tutorial is fully decentralized, and is based on a distributed public-key infrastructure similar to SDSI [27]. The model supports peer-to-peer delegation of authority under fine-grained caveats [10]. Access control policies are based on human-readable names, and have support for negative clauses and group-based checks. This general purpose model is applicable in various distributed system settings, including, peer-to-peer computing environments, IoT, cloud, and enterprise. The model has been deployed as part of an open-source application framework called *Vanadium* [6], and is thus referred to as the Vanadium authorization model. As part of this tutorial, we survey a number of key ideas and techniques from previous work on distributed authorization, including SPKI/SDSI [18,27], Macaroons [10], and the vast body of work on trust management [12]. We explain how these techniques are combined by the Vanadium authorization model.

Organization. The rest of this tutorial is organized as follows. Section 2 describes the features we desire, and Sect. 3 surveys key technical ideas involved in the design of our model. Section 4 describes our model in detail, followed by an application of our model to a physical lock device in Sect. 5. Section 6 concludes.

2 Desired Features

In this section, we describe features we seek from our authorization model. Many of these features have been considered by prior work on distributed authorization. We use the term "principal" informally to refer to an entity in our system, including, users, devices, processes, and objects, and leave the formal definition to Sect. 4.

Decentralization. As discussed in the introduction, decentralized deployment and peer-to-peer communication by default are the central guiding principles behind this work. The model must not force dependence on special global authorities such as x.509 certification authorities, default identity providers, and proxies that mediate interactions between principals. Instead, we seek egalitarian systems where *any* principal can be an authority for some set of other principals. For instance, a user Alice may become an authority on the identities and access controls on all her home devices. The devices may be configured to specifically trust only Alice's credentials. In general, devices must be able to securely communicate with each other as long as there is a direct communication channel between them. We seek a model that minimizes interaction with globally accessible services, and maximizes what can be achieved with direct peer-to-peer communication.

Mutual Authentication and Authorization. We require all interactions between principals to be *mutually* authenticated and authorized. The principal at each end of a communication channel must identify the principal at the other end (authentication), and verify that it is valid in the context of the communication (authorization). Mutual authentication is essential for both ends to

audit all access; we discuss the benefit of auditing later in this section. Mutual authorization is essential whenever the communicating principals are mutually suspicious. Unidirectional authorization often opens the door to rogue entities, leading to security and privacy attacks. For example, when Bob tries to unlock the lock on Alice's front door, the lock must be convinced that it is communicating with Bob, and that Bob is authorized to unlock the door. At the same time, Bob must be convinced that it is indeed sending the request to Alice's front door, and not an imposter device that is tracking Bob's behavior.

Compound Identities. We live in a world today where all users and devices carry multiple identities. For instance, a user may have an identity from social media sites (e.g., Facebook identity), an identity from her work place, an identity from the government (e.g., passport, driver's license), and so on. Similarly, a device would have an identity from the manufacturer (e.g., Samsung TV model 123), and an identity from the device owner (e.g., Alice's TV). A principal must be able to act under one or more identities associated with it, and different identities may grant different authorizations to the principal. The authorization model must seamlessly capture this compound nature of a principal's identity.

Fine-Grained Delegation. The strength of distributed systems lies in multiple computing agents coming together to accomplish complex tasks. This is indeed the promise of IoT. For example, Bob would like to play a movie from his internet video service on Alice's TV and speaker system. In order to enable such interactions, the authorization model must support flexible sharing and delegation.[1] Alice must be able to delegate access to her TV to Bob, and Bob must be able to delegate access to his internet video service to Alice's TV for playing a particular movie. The model must also support delegations across multiple hops. For instance, Bob must be able to easily delegate access to Alice's TV to his friend Carol. Moreover, in light of the decentralization requirement, we would like delegations to work peer-to-peer rather than be mediated by a central authority.

In practice, delegation of authority is seldom unconditional, and thus the delegation mechanism must support constraints on the scope of the delegation. For instance, Bob may want Alice's TV to have access to his internet video service only for playing a particular movie, and only while Bob is present in Alice's house. The TV must loose access as soon as Bob leaves the house. Alice may want the same for Bob's access to her TV.

We emphasize that the delegation mechanism must be flexible and convenient to use. Inflexibilities or inconveniences in the mechanism not only affect the user experience but are also detrimental to security as they push users to look for

[1] Some systems choose to distinguish the concepts of "sharing" an "delegation" with the former being a mechanism for a principal to allow another principal to access an object while the latter being a mechanism for allowing another principal to act on its behalf. In this tutorial, we do not make this distinction, and treat "delegation" more broadly as a mechanism for one principal to delegate some of its authority to another principal.

insecure workarounds. For instance, in the absence of a convenient mechanism to share access to an internet video service, Bob may end up sharing his account password with the TV, and as a result give away access to all his account data (e.g., viewing history, purchases) instead of just access to a particular movie.

Auditable Access. In a system with delegation, users should be able to audit the use of delegated access over time. For instance, a user must be able to determine who has access to her devices and who has *exercised* that access. Delegations are ultimately tied to the intention of the human end-user, and software must acknowledge that it is impossible to codify all possible human intentions. This is particularly true when users themselves may be unable to clearly articulate their intentions at the time of delegation. An accurate audit trail is a requirement to detect mismatches between user intentions and their codification. For example, Alice might give Charlie access to her home to come by and walk her dog once a day. Alice's intent is for Charlie to be a dog walker but she cannot possibly know apriori what time Charlie will come by in all future days. Auditing Charlie's use of the authority granted to her by Alice is necessary to detect a violation of the contract between the two. Moreover, this auditing must be fine grained—if Charlie was tricked into running a malicious application on her phone, Alice must be able to pinpoint the exact application that was improperly using the authority she granted to Charlie.

Revocation. Users make mistakes, devices get stolen/compromised, and relationships break. As a result an authorization model must support revocation. For instance, Alice must be able revoke Dave's access to all her home devices when they have a falling out. Similarly, she should be able to revoke all delegations that she made to her tablet when her tablet gets stolen.

Ease of Use. Usability is a key determining factor in the effectiveness of security systems [14,30]. Systems with complex interfaces and mechanisms often have degraded security, as users tend to look for insecure workarounds when dealing with them. Thus we strive to design an authorization model that can be easily understood and configured by system developers, and lends itself well to simple and clear user interfaces.

3 Background

Distributed authorization is a very mature field with decades of prior research. The Vanadium authorization model is a result of combining various known techniques in order to meet the requirements stated in the previous section. In this section, we provide some background on distributed authorization, and discuss the key ideas involved in the design of the Vanadium authorization model.

In essence, most authorization mechanisms involve a requester presenting a set of credentials (possibly obtained from multiple parties) to a reference monitor guarding a resource which then authorizes the request after validating the credentials [29]. A common paradigm is for the requester to present credentials

that establish its identity, which is then checked against an access control policy (e.g., an access control list (ACL)) by the reference monitor. Various mechanisms differ in the type of credentials involved. In mechanisms such as OAuth2 [21], OpenID [3], Macaroons [10], and many others [20,25], the credentials are essentially tokens constructed using symmetric-key cryptography by an issuer (e.g., an identity provider in the case of OAuth2). These mechanisms are simple, efficient, easy to deploy, and are widely in use (particularly OAuth2) for client authorization on the Web. The downside is that the credentials can be validated and interpreted only by the credential issuer. As a result, the credential issuer must be invoked for validating credentials during each request[2]. This is undesirable in our setting.

In contrast, mechanisms based on public-key certificates [9,11–13,23,27,32] do not suffer from this downside. In these mechanisms, principals possess digital signature public and secret key pairs along with signed certificates binding authorizations to their public key. A principal makes requests by signing statements using its secret key and presenting one or more of its certificates. These certificates can be validated and interpreted by any principal as long as it knows the public key of the certificate issuer. Such a mechanism is used for authenticating HTTPS [17] servers on the Web.

Certificate-based mechanisms rely on a *public-key infrastructure* (PKI) for distributing certificates to various principals. Traditional x509 PKI [28], such as the one used on the Web, is centralized with a hierarchical network of certification authorities responsible for issuing certificates. In light of the downsides of centralization, several decentralized PKI [11,12,18,27,32] have been proposed in the literature. A prominent model among these is the simple distributed security infrastructure (SDSI) [27] of Rivest and Lampson. The SDSI model was subsequently merged with the simple public key infrastructure (SPKI) effort [18], and the resulting model is commonly referred to as SPKI/SDSI. In what follows, we briefly summarize some of the key ideas in SPKI/SDSI, while referring the reader to [18,27] for a more comprehensive description.

SPKI/SDSI. This is a decentralized PKI based on the idea of *local names*. Each principal in this model is represented by a digital signature public key, and manages a local name for referring to other principals. For instance, a principal Alice may use the name `friend` to refer to her friend Bob's public key and `doctor` to refer to her family doctor Frank's public key. These bindings are local to Alice, and other principals may chose to bind different names to these keys. However, another principal, say Alice's TV, who names Alice's public key as `Alice` may refer to Bob's public key as `Alice's friend`. Thus names in different namespaces can be linked using the `'s` operator. Local name to key bindings are represented by *name-definition* certificates signed by the issuing principal. Linked names are thus realized by certificate chains. The model also supports *authorization* certificates wherein an issuing principal delegates permissions to another principal.

[2] An alternative is for each resource owner to become a credential issuer but that leads to a proliferation of credentials at the requester's end.

Access control policies in SPKI/SDSI specify a list of authorized principals using local names in the owner's namespace. A request is allowed by the policy if the requesting principal is directly authorized by the policy or has a delegation (via authorization certificate) from a directly authorized principal. For instance, Alice's TV may have an access control policy allowing the local name `Alice's friend`, and therefore Bob (who has the name `Alice's friend` in the TV's namespace) and any principal delegated by Bob will have access. Authorizing requests involves assembling a chain of certificates (from a repository of certificates) that proves that the requesting principal satisfies the access control policy [16]. The responsibility of assembling the right certificate chain may be placed on the reference monitor or the requester, and various deployments may differ in this choice. The key idea from SPKI/SDSI used in the Vanadium authorization model is that of delegating access to principals by assigning them local names, and basing access control policies on these names.

Caveats on Delegation. There are several mechanisms in the literature on restricting the scope of delegations. These range from simple, coarse-grained mechanisms of adding a purpose and expiration time to delegation certificates, to complex, fine-grained mechanisms of extending delegation certificates with S-expressions capturing application-specific permissions [18], or program code [13] defining how access must be proxied to the resource. Recently, Birgisson et al., proposed a mechanism for restricting delegations using *caveats* [10], which aims at striking a balance between simplicity and expressiveness.

Caveats are essentially predicates that restrict the context in which the delegated credential may be used. They are attached to the credential in a tamper-proof manner, each time the credential is delegated. Caveats are of two types—*first-party* and *third-party*. First-party caveats are predicates on the context in which a credential may be used. For e.g., first-party caveats impose restrictions on the time of request (e.g., only between 6 PM to 9 PM), the permitted operation (e.g., only Read requests), the requester's IP address (e.g., the IP address must not be blacklisted), etc. These restrictions are validated by the reference monitor in the context of an incoming request, and the credential is considered invalid if any caveat present on it is invalid.

Third-party caveats are restrictions wherein the burden of validation is pushed to a third-party, i.e., neither the party that wields the credential nor the party that is authorizing it. A credential with a third-party caveat is considered valid only when accompanied by a *discharge* (proof of validity) issued by the specific third party mentioned in the caveat. This discharge must be obtained by the holder of the credential before using the credential as part of a request. A reference monitor making authorization decisions simply checks that valid discharges are provided for all third-party caveats on the credential.

A third-party caveat can be used for implementing revocation checks by having the discharge service issue discharges only if the credential has not been revoked. The discharge may be short-lived, and thus the holder of the credential would be obligated to periodically obtain fresh discharges from the service. Although this mechanism seems similar to the online certificate status protocol

(OCSP) [24], the key difference is that unlike an OCSP response, fetching a discharge is the responsibility of the principal making the request rather than the one authorizing it. Other examples of third-party caveats would be restrictions pointed at a social networking service (e.g., discharged by checking membership in the "work" circle), or an auditing service (e.g., discharged by adding an entry to the audit log). Discharges may themselves carry first-party and third-party caveats thereby making the overall mechanism highly expressive. For instance, a *parental-control* caveat on a credential handed to a kid may initially point to a service on dad's phone who in some cases may issue a discharge with a third-party caveat pointed at mom's phone. While caveats were originally designed in the context of Macaroons [10], in this work we use them to restrict the scope of delegation certificates in Vanadium.

4 Vanadium Authorization Model

In this section, we describe the authorization model of the *Vanadium* framework [6]. Vanadium is a set of tools, libraries, and services for developing secure distributed applications that can run over a network of devices. At the core of the framework is a remote procedure call (RPC) system that enables applications to expose services over the network. The framework offers an interface definition language (IDL) for defining services, a federated naming system for addressing them, and an API for discovering accessible services. The authorization model is responsible for controlling access to RPC services, and ensuring that all RPCs are end-to-end encrypted, mutually authenticated, and mutually authorized.

Preliminaries. The model makes use of a digital signature scheme (e.g., ECDSA P-256). In particular, we assume public and secret key pairs (pk, sk), and algorithms sign and verify for signing messages and verifying signatures respectively. $\mathsf{sign}(sk, msg)$ uses a secret key sk to produce a signature over a message msg, and $\mathsf{verify}(pk, msg, sig)$ verifies a signature sig over a message msg using a public key pk. For any public and secret key pairs pk, sk, $\forall msg : \mathsf{verify}(pk, msg, \mathsf{sign}(sk, msg))$ holds. We also assume a cryptographically secure hash function (e.g., SHA256), denoted by hash. For convenience, we assume that hash takes an arbitrary number of arguments of arbitrary type, and internally encodes all arguments into a byte array using some lossless encoding technique.

4.1 Principals and Blessings

A principal is any entity that can interact in the Vanadium framework. Specifically, processes, applications, and services that include a Vanadium runtime are all principals. Each principal is associated with a public and secret key pair, with the secret key never being shared over the network. Each principal has one or more hierarchical human-readable names associated with it called *blessings*.

For instance, a television set (TV) owned by a user Alice[3] may have a blessing `Alice / TV`. Principals can have multiple blessings, and thus the same TV may also have a blessing `PopularCorp / TV123` from its manufacturer.

Principals are authenticated and authorized during requests based on the blessing names bound to them. The public key of the principal does not matter as long as the principal can prove that it has a blessing name satisfying the other end's access control policy. We believe that this choice makes it easier and more natural for users and system administrators to define access control policies and inspect audit trails as they have to reason only in terms of human-readable names. Concretely, a blessing is represented via a chain of certificates. The formal definition of certificates and blessings is given in Fig. 1. We use $\langle \ldots \rangle$ to define tuples, colon as a binary operator for forming lists, and `empty` for the empty list. n ranges over ordinary names not containing `/`. Certificates, denoted by C, contain exactly four fields—a name, a public key, a (possibly empty) list of caveats, and a digital signature. We discuss the definition of caveats a bit later; for now they can be thought of as restrictions (e.g., expiration time) on the validity of the certificate. We use the dot notation to refer to fields of a tuple, and thus $C.n$ is the name of a certificate C. Notice that our certificate format is much simpler in contrast to x509 certificates [28].

$$
\begin{array}{rcll}
n & ::= & \text{ordinary names not containing } \texttt{/} & \text{names} \\
C & ::= & \textsc{c}\langle n, pk, clist, sig\rangle & \text{certificates} \\
B & ::= & C & \text{blessings} \\
 & | & B : C & \\
clist & ::= & \texttt{empty} & \text{list of caveats} \\
 & | & clist : c & \\
 & & & \\
c & ::= & fc \mid tc & \text{caveats} \\
fc & ::= & \mathsf{timeCaveat} & \text{first-party caveats} \\
 & | & \mathsf{targetCaveat} & \\
 & | & \ldots & \\
tc & ::= & \textsc{t}\langle nonce, pk, fc, loc\rangle & \text{third-party caveats} \\
d & ::= & \textsc{d}\langle clist, sig\rangle & \text{discharges}
\end{array}
$$

Fig. 1. Certificates, blessings and caveats

Blessings (denoted by B) are non-empty lists of certificates with each certificate capturing a component of the blessing name. The list of certificates is meant to form a chain such that signature of each certificate except the first one can be verified using the public key of the previous certificate. The first certificate is self-signed, that is, its signature can be verified by its own public key. The public key listed in the final certificate is the public key of the principal to which

[3] For ease of discussion, we refer to users and devices as principals; strictly speaking, we are referring to processes controlled by them.

the blessing is bound. This public key is denoted by $\mathsf{pk}(B)$ for a blessing B. The name of a blessing is obtained by concatenating all names appearing in the blessing's certificate chain using `/`. This name is denoted by $\mathsf{nm}(B)$.

As an example, the blessing `PopularCorp / TV123` is bound to the television's public key pk_{TV} by a chain of two certificates—(1) a certificate with name `PopularCorp` and public key $pk_{PopularCorp}$, signed by $sk_{PopularCorp}$, and (2) a certificate with name `TV123` and public key pk_{TV}, signed by $sk_{PopularCorp}$. The validity of the certificate chain of a blessing B is defined by the predicate $\mathsf{IsValidChain}(B)$.

$$
\begin{array}{l}
\mathsf{IsValidChain}(B) ::= \\
\quad \texttt{case}\ B\ \texttt{of} \\
\qquad C \quad\ : \mathsf{verify}(C.pk, \mathsf{hash}(C.n, C.pk, C.clist), C.sig) \\
\qquad \mid\ B{:}C : \mathsf{IsValidChain}(B) \wedge \mathsf{verify}(\mathsf{pk}(B), \mathsf{hash}(B, C.n, C.pk, C.clist), C.sig)
\end{array}
$$

The signature in each certificate of the chain is not just over the certificate's contents but also the chain leading up to the certificate. This means that each certificate is cryptographically integrated into the blessing, and cannot be extracted and used in another blessing. This property does not hold for SPKI/SDSI and many other certificate-based systems, where certificates can be chained together in arbitrary ways. As a result, Vanadium does not face the *certificate chain discovery* problem [16] that involves assembling the right chain of certificates (from a repository) to prove that certain credentials are associated with a public key. A Vanadium blessing is a tightly bound certificate chain containing all the certificates required to prove that a certain name is bound to a public key.

4.2 Delegation and Caveats

Blessings can be delegated from one principal to another by extending them with additional names. For instance, Alice may extend her blessing `Alice` to her TV as `Alice / TV`. The TV may in turn extend this blessing to the Youtube application running on it as `Alice / TV /Youtube`. Since principals possess authority by virtue of their blessing names, delegating a blessing amounts to delegation of authority. For instance, delegation of the blessing `Alice / TV` allows the TV to access all resources protected by an ACL of the form `Allow Alice / TV`.

Concretely, a delegation is carried out by the operation $\mathsf{Bless}(pk, sk, B, n, clist)$ which takes the public key pk_d of the delegate, the secret key sk of the delegator, the blessing B that must be extended, the name n used for the extension, and the list of caveats $clist$ on the delegation. It extends the blessing's certificate chain with a certificate containing name n, public key pk_d, list of caveats $clist$, and signed by the secret key sk.

$$\mathsf{Bless}(pk_d, sk, B, n, clist) ::= B{:}\mathrm{C}\langle n, pk_d, clist, \mathsf{sign}(sk, \mathsf{hash}(B, n, pk, clist))\rangle$$

It is easy to see that if the blessing B is valid, and the secret key sk corresponds to the public key $\mathsf{pk}(B)$, then the blessing $\mathsf{Bless}(pk_d, sk, B, n, clist)$ is valid. Each blessing delegation involves the blesser choosing an extension (e.g.,

`TV`) for the blessing. The extension may itself have multiple components (e.g., `home / bedroom / TV`), and a blesser may choose the same or a different extension across multiple delegations. The role of the extension is to namespace the delegation, similar to *local names* in SDSI [27].

The role of the caveat list (*clist*) supplied to the Bless operation is to restrict the conditions under which the resulting blessing can be used. For example, Alice can bless her TV as `Alice / TV` but with the caveats that the blessing can be used only between `6 PM` and `9 PM`, and only to make requests to her video service (and not her bank!). Thus, the resulting blessing makes the assertion:

> The name `Alice:TV` is bound to pk_{TV}
> *as long as* the TIME is between `6 PM` and `9 PM`, and
> *as long as* target of the request matches `SomeCorp / VideoService`

When the TV uses this blessing to make a request to the video service, the service will grant the request only if the current time is within the permitted range, and its own blessing name matches `SomeCorp / VideoService`.

The caveats in the above example are all first-party caveats as they are validated by the reference monitor at the target service when the blessing is presented during a request. As discussed in Sect. 3 and [10], first-party caveats are validated in the *context* of an incoming request based on information such as the time of request, the blessing presented, the method invoked, etc. We use $\mathcal{C}$ to denote request contexts, and assume a function $\mathsf{IsValidFCav}(fc, \mathcal{C})$ that checks if a first-party caveat fc is valid in the context $\mathcal{C}$.

While the Vanadium framework implements IsValidFCav on some standard first-party caveats (e.g., expiry, method and peer restriction), services may also define their own first-party caveats. For instance, a video streaming service may define a "PG-13" caveat so that blessings carrying this caveat would be authorized to stream only PG-13 movies.

4.3 Third-Party Caveats

Vanadium blessings may also carry third-party caveats [10] wherein the burden of validating the caveat is pushed to a third-party. For example, Alice can bless Bob as `Alice / Houseguest / Bob` but with the caveat that the blessing is valid only if Bob is within 20 feet of Alice as determined by a proximity service running on Alice's phone (third-party). Before making a request with this blessing, Bob must obtain a *discharge* (proof) from the proximity service on Alice's phone. This discharge must be sent along with the blessing in a request. Thus, the blessing makes the assertion:

> The name `Alice / Houseguest / Bob` is bound to pk_{Bob}
> *as long as* a proximity service on Alice's phone issues a discharge after validating that Bob is "within 20 feet" of it.

The structure of a third-party caveat and discharge is defined in Fig. 1. Every third-party caveat includes a nonce (for uniqueness), a public key controlled by

the third-party service, a first-party caveat specifying the check that must carried out before issuing the discharge, and the location of the third-party service. A discharge contains a signature and possibly additional caveats. It is considered valid if its signature can be verified by the public key listed in the third-party caveat, and any additional caveats listed on it are valid.

The operation $\mathsf{MintDischarge}(sk_{tp}, tc, clist, \mathcal{C})$ uses a secret key sk_{tp} (owned by the third-party) to produce a discharge for a third-party caveat tc if the check specified in tc is valid in the context $\mathcal{C}$. The returned discharge contains the provided list of caveats $clist$.

$$
\begin{array}{l}
\mathsf{MintDischarge}(sk_{tp}, tc, clist, \mathcal{C}) ::= \\
\quad \mathtt{case}\ \mathsf{IsValidFCav}(tc.fc, \mathcal{C})\ \mathtt{of} \\
\qquad \mathtt{true}\ \ : \mathtt{error} \\
\qquad \mid\ \mathtt{false} : \textsc{d}\langle clist, \mathsf{sign}(sk_{tp}, \mathsf{hash}(tc, clist))\rangle
\end{array}
$$

The purpose of the additional caveats on the discharge is to limit its validity. For instance, in the proximity caveat example, Alice's phone may issue a discharge that expires in 5 min thereby requiring `Bob` to fetch a new discharge periodically. This ensures that Bob cannot cheat by fetching a discharge when near Alice's phone, and then using it later from a different location.

4.4 Blessing Roots

Since blessing names are the basis of authorization, it is important for them to be unforgeable. Simply verifying the signatures in a certificate chain does not protect against forgery because the first certificate in the chain is self-signed, and thus can be constructed by any principal. For instance, an attacker with public and secret keys pk_a, sk_a can bind the name `Alice` to her public key using the blessing $\textsc{c}\langle \mathtt{Alice}, pk_a, \mathtt{empty}, \mathsf{sign}(sk_a, \mathsf{hash}(\mathtt{Alice}, pk_a, \mathtt{empty}))\rangle$. She may then extend this blessing with any extension of her choosing using the Bless operation.

In order to defend against such forgery, all Vanadium principals have a set of *recognized roots*. The *root* of a blessing is the public key and name of the first certificate in the blessing's certificate chain. A blessing is recognized by a principal only if the blessing's root is recognized. For instance, all of Alice's devices may recognize her public key pk_{Alice} and name `Alice` as a root. Any blessing prefixed with `Alice` would be recognized only if it has pk_{Alice} as the root public key. Similarly all devices manufactured by PopularCorp may recognize the public key $pk_{PopularCorp}$ and name `PopularCorp` as a root. Devices from another manufacturer may not recognize this root, and would simply discard blessings from PopularCorp.

We use the term *identity provider* for a principal that acts as a blessing root and hands blessings to other principals. For instance, both Alice and PopularCorp would be considered as identity providers by Alice's TV. Any principal can become an identity provider. In general, we anticipate well-known companies, schools, and public agencies to become identity providers, and different principals may recognize different subsets of these.

4.5 Authentication and Authorization

Client and servers in a remote procedure call (RPC) authenticate each other by presenting blessings bound to their public keys. The Vanadium authentication protocol [7] is based on the well-known SIGMA-I protocol [15]. It ensures that each end learns the other end's blessing, and is convinced that the other end possesses the corresponding secret key. At the end of the protocol, an encrypted channel (based on a shared key) is established between the client and server for further communication. Since the protocol is fairly standard, we do not discuss it here and refer the reader to [7] for a detailed description.

Once authentication completes, each end checks that the other end's blessing is authorized for the RPC. Authorization is mutual. For example, Alice (client) may invoke a method on her TV (server) only if the TV presents a blessing matching `Alice / TV`, and the TV may authorize Alice's request only if she presents a blessing prefixed with `Alice`. Authorization checks involve two key steps—(1) validating the blessing presented by the other end, followed by (2) matching the blessing name against an access control policy.

A blessing is always validated in the context of a given request. Blessing validation involves —(1) verifying the signatures on the blessing's certificate chain, (2) verifying that the blessing root is recognized, and (3) validating all caveats on the blessing in the context of the request.

$\mathsf{IsValidBlessing}(B, \mathcal{R}, \mathcal{C})$, defined below, determines if a blessing B is valid for a given request context $\mathcal{C}$ and a set of recognized roots $\mathcal{R}$. The context is assumed to contain all parameters of the request, including any discharges sent by the other end. Specifically, $\mathsf{dis}(\mathcal{C})$ denotes the discharges contained in the context $\mathcal{C}$, $\mathsf{root}(B)$ is the root of the blessing B and $\mathsf{cavs}(B)$ is the set of all caveats appearing on certificates of the blessing B. A first-party caveat is validated by invoking the function $\mathsf{IsValidFCav}$, and a third-party caveat is validated by finding a matching discharge in the request context, i.e., a discharge whose signature can be verified using the public key specified in the third-party caveat. Additionally, any caveats listed on the discharge are also (recursively) validated.

$$\mathsf{IsValidBlessing}(B, \mathcal{R}, \mathcal{C}) ::= \mathsf{IsValidChain}(B) \wedge \mathsf{IsRecognized}(B, \mathcal{R}) \wedge \mathsf{IsValidCavs}(\mathsf{cavs}(B), \mathcal{C})$$

$$\mathsf{IsRecognized}(B, \mathcal{R}) ::= \exists r \in \mathcal{R} : \mathsf{root}(B) = r$$

$$\begin{array}{l}\mathsf{IsValidCavs}(\mathit{clist}, \mathcal{C}) ::= \\ \quad \forall c \in \mathit{clist} : \\ \quad\quad \texttt{case } c \texttt{ of} \\ \quad\quad\quad \mathit{fc} \ : \mathsf{IsValidFCav}(\mathit{fc}, \mathcal{C}) \\ \quad\quad\quad \mid \ \mathit{tc} : \exists d \in \mathsf{dis}(\mathcal{C}) : \\ \quad\quad\quad\quad\quad \mathsf{verify}(\mathit{tc}.\mathit{pk}, \mathsf{hash}(\mathit{tc}, d.\mathit{clist}), d.\mathit{sig})) \wedge \mathsf{IsValidCavs}(d.\mathit{clist}, \mathcal{C})\end{array}$$

Once the remote end's blessing is validated, the name of the blessing is checked against an access control policy. Access is granted only if the check is successful. We discuss the structure of these policies next.

4.6 Access Control Policies

The syntax and semantics of access control policies in Vanadium has been defined rigorously in [8]. We give a brief overview of the design in this subsection. Policies in Vanadium are specified using access control lists (ACLs) that resolve to the set of permitted blessing names. In order to allow policies to be short and simple, Vanadium allows ACLs to indirect through groups. For example, Alice may create a group $\texttt{AliceFriends}_{\texttt{G}}$ containing the blessing names of all her friends and add it to all relevant ACLs. This saves her from enumerating the list of blessing names of family members within each ACL, and also provides a central place to manage multiple policies. Furthermore, group definitions may be nested. For instance, the definition of the group $\texttt{AliceFriends}_{\texttt{G}}$ may contain the group $\texttt{DaveFriends}_{\texttt{G}}$ containing the friends of Alice's flatmate Dave. Typically, the definition of a group would be held at a remote server, which would be contacted during ACL resolution. The ACL resolver may cache information about groups in order to combat unreliable network connectivity and avoid expensive network roundtrips.

$$
\begin{array}{rll}
bn ::= & n & \text{blessing names} \\
& \mid n \texttt{ / } bn & \\
p ::= & n & \text{blessing patterns} \\
& \mid g & \\
& \mid n \texttt{ / } p & \\
& \mid g \texttt{ / } p & \\
P ::= & \texttt{empty} & \text{list of blessing patterns} \\
& \mid P \texttt{:} p & \\
A ::= & \texttt{Allow } P \texttt{ Deny } P & \text{ACL}
\end{array}
$$

Fig. 2. Access control policies

ACLs may also contain blessing names where one of the components is a group name. Such names are called *blessing patterns*, and are meant to capture a derived set of blessing names. For example, the pattern $\texttt{AliceFriends}_{\texttt{G}}\texttt{ / Phone}$ defines the set of blessing names of phones of Alice's friends. In particular, if the group $\texttt{AliceFriends}_{\texttt{G}}$ contains the blessing name `Bob`, then the pattern $\texttt{AliceFriends}_{\texttt{G}}\texttt{ / Phone}$ would be matched by the blessing name `Bob / Phone`.

Definitions. The formal definition of blessing patterns and ACLs is given in Fig. 2. As before, n ranges over ordinary names not including `/`. g ranges over group names (i.e., name with the subscript "`G`"), and bn ranges over blessing names. A blessing pattern is a non-empty sequence of ordinary names or group names separated by `/`. An ACL is a pair of `Allow` and `Deny` clauses, each containing a list of blessing patterns.[4] `Deny` clauses make it convenient to encode blacklists. For instance, the ACL $\texttt{Allow AliceDevices}_{\texttt{G}}\texttt{ Deny AliceWorkDevices}_{\texttt{G}}$ allows access to all of Alice's devices except her work devices.

[4] The model described in [8] is more general and allows multiple `Allow` and `Deny` clauses in ACLs.

Group definitions are of the form $g =_{\text{def}} P$, and thus equate a group name with a list of blessing patterns. For example, the group $\texttt{AliceFriends}_\texttt{G}$ may have a definition of the form

$$\texttt{AliceFriends}_\texttt{G} =_{\text{def}} \texttt{Bob}, \texttt{Carol}, \texttt{DaveFriends}_\texttt{G}$$

A given blessing name satisfies an ACL if there is at least one blessing pattern in the `Allow` clause that is matched by the blessing name, and no blessing pattern in the `Deny` clause is matched by the blessing name. For example, when `Bob` is in the group $\texttt{AliceFriends}_\texttt{G}$, the ACL $\texttt{Allow AliceFriends}_\texttt{G}$ will permit access with the blessing name `Bob` but the ACL $\texttt{Allow AliceFriends}_\texttt{G}\ \texttt{Deny Bob}$ will deny it. The default is to deny access, so for example the ACL `Allow Bob` will deny access to `Carol`.

The semantics of ACL checks makes use of the prefix relation on blessing names. Given two blessing names bn_1 and bn_2, we write $bn_1 \preceq bn_2$ if the sequence of names in bn_1 (separated by `/`) is a prefix of the sequence of names in bn_2, for e.g., $\texttt{Alice} \preceq \texttt{Alice/TV}$ holds but $\texttt{Ali} \preceq \texttt{Alice/TV}$ does not.

In order to formalize the ACL checking procedure, we first define a function $\mathsf{Meaning}_\rho$ that maps a blessing pattern to a set of blessing names. It is parametric on a semantics of group names, which is a function ρ that maps a group name to a set of members of the group. We discuss how ρ is obtained later.

$$\begin{array}{l}
\mathsf{Meaning}_\rho(p) ::= \\
\quad \texttt{case } p \texttt{ of} \\
\qquad\quad n \quad\quad : \{n\} \\
\qquad \mid\ g \quad\quad : \rho(g) \\
\qquad \mid\ n \texttt{ / } p : \{n \texttt{ / } s \mid s \in \mathsf{Meaning}_\rho(p)\} \\
\qquad \mid\ g \texttt{ / } p : \{s \texttt{ / } s' \mid s \in \mathsf{Meaning}_\rho(g), s' \in \mathsf{Meaning}_\rho(p)\}
\end{array}$$

$\mathsf{Meaning}_\rho$ can be naturally extended to a list of blessing patterns by defining it as the union of the sets obtained by applying $\mathsf{Meaning}_\rho$ to each element of the list, with $\mathsf{Meaning}_\rho(\texttt{empty})$ defined as $\emptyset$. Using the function $\mathsf{Meaning}_\rho$, we define the function $\mathsf{IsAuthorized}(bn, A)$ that decides whether a given blessing name bn (seen during a request) satisfies an ACL A.

$$\begin{array}{l}
\mathsf{IsAuthorized}(bn, A) ::= \\
\quad \texttt{case } A \texttt{ of} \\
\qquad \texttt{Allow } P_A \ \ \texttt{Deny } P_D : (\exists bn' \in \mathsf{Meaning}_\rho(P_A).bn' \preceq bn) \\
\qquad\qquad\qquad\qquad \wedge (\neg \exists bn' \in \mathsf{Meaning}_\rho(P_D).bn' \preceq bn)
\end{array}$$

The function checks that the blessing name bn matches an allowed blessing pattern and does not match any denied blessing pattern. Matching is defined using prefixes (instead of exact equality) for both allowed and denied clauses; the reasons however are different.

For `Allow` clauses, the use of the relation $\preceq$ is a matter of convenience. We believe that often when a service grant access to a principal (e.g., with blessing name `Alice`) it may be fine if the access is exercised by delegates of the principal

(e.g., with blessing name `Alice / Phone`). Thus a pattern in an `Allow` clause is considered matched as long as some prefix of the provided blessing name matches it. Alternatively, services that want to force exact matching for allowed patterns—perhaps to prevent the granted access from naturally flowing over to delegates—may use the special reserved name *eob* at the end of the pattern. For e.g., the pattern `Allow Alice / eob` is matched only by the blessing name `Alice`.

For `Deny` clauses, ensuring that no prefix of the blessing name matches a denied pattern is crucial for security. A principal with blessing name bn can always extend it (using the Bless operation) and bind it to itself. Thus, from a security perspective it is important that if bn is denied access, then all extensions of bn are also denied access.

Semantics of groups (ρ). We now discuss how the map ρ from group names to members of the group is defined. In Vanadium, groups may also be distributed, in that, different group definitions may be held at different servers. Given this, defining the map ρ becomes complicated for several reasons. Firstly, due to network partitions some group servers may be unreachable during ACL checking and thus their definitions may be unavailable. The definition of a group may depend on other groups, and there may be no overseeing authority ensuring absence of dependency cycles. Finally, group server checks need to be secure and private. For instance, group servers under different administrative domains may be unwilling to reveal their complete membership lists to each other, and may offer only an interface for membership lookups. We refer the reader to [8] for a complete treatment of how these issues are tackled, and explain only the key ideas below.

When group servers are unreachable during ACL checking, we conservatively approximate the definition of ρ for those groups. The approximation depends on whether the group is being resolved in the context of an `Allow` clause or a `Deny` clause. While matching an allowed pattern, unreachable groups are under-approximated by the empty set. On the other hand, while matching a denied pattern, unreachable groups are over-approximated by the set of all blessings. Thus effectively we define two maps $\rho_{\Downarrow}$ and $\rho_{\Uparrow}$—the map $\rho_{\Downarrow}$ is used while defining $\mathsf{Meaning}_{\rho}$ for allowed patterns, and the map $\rho_{\Uparrow}$ is used while defining $\mathsf{Meaning}_{\rho}$ for denied patterns. The maps $\rho_{\Downarrow}$ and $\rho_{\Uparrow}$ coincide when all group definitions are available.

$\rho_{\Downarrow}$ and $\rho_{\Uparrow}$ can be constructed by considering the list of available group definitions as a set of productions inducing a formal language. Ordinary names and `/` are terminals, and group names are non-terminals. For instance, the group definition

$$g_1 =_{\text{def}} \texttt{Alice / Phone},\ g_2\ \texttt{/ Phone}$$

can be viewed as two productions

$$g_1 \rightarrow \texttt{Alice / Phone}$$
$$g_1 \rightarrow g_2\ \texttt{/ Phone}$$

For $\rho_{\Downarrow}$, no production is associated with a group name whose definition is unavailable. On the other hand for $\rho_{\Uparrow}$, such group names are associated with productions inducing the set of all blessings. Once the set of productions are defined, for any group name g, $\rho_{\Downarrow}(g)$ and $\rho_{\Uparrow}(g)$ are defined as the set of blessing names generated from g by the corresponding set of productions.

While constructing $\rho_{\Downarrow}$ and $\rho_{\Uparrow}$ in the aforesaid manner is infeasible in practice as it requires knowledge of all the group definitions, the key observation here is that checking group membership can be reduced to checking membership in an induced formal language. In [8], this observation and techniques from top-down parsing are used to develop a distributed algorithm for checking whether a blessing name belongs to a group.

4.7 Life of an RPC

We now explain how the various parts of the authorization model come together during the course of an RPC. Consider Alice's house guest Bob who wants to invoke a method on Alice's TV. Suppose Bob has the blessing B_{Bob} with name `Alice / Houseguest / Bob`, and the TV has the blessing B_{TV} with name `Alice / TV`. Additionally, suppose that Bob's blessing has a third-party caveat to a proximity service running on Alice's phone. Bob has the access control policy `Allow Alice / TV` that allows only Alice's TV, and the TV has the access control policy `Allow Alice:Alice / Houseguest` that allows only Alice and her house guests. In what follows, we describe the steps involved in the RPC from Bob's phone to Alice's TV. We focus on the authentication and access-control aspects, and do not discuss how various network connections are established.

- Bob uses the Vanadium authentication protocol to initiate a connection to Alice's TV.
- As part of the exchange, the TV first sends its blessing B_{TV} to Bob. Bob invokes $\mathsf{IsValidBlessing}(B_{TV}, \mathcal{R}_{Bob}, \mathcal{C}_{Bob})$ to verify that the blessing B_{TV} is valid for the current context from Bob's perspective and the set of blessing roots recognized by Bob ($\mathcal{R}_{Bob}$). If this step succeeds, then Bob verifies that the name of the blessing (`Alice / TV`) satisfies his access control policy (`Allow Alice / TV`). The connection is aborted if any of these checks fail.
- After authorizing the TV's blessing, Bob selects his blessing B_{Bob} (from Alice) to present to the TV. Since the blessing carries a third-party caveat, Bob first connects to the third-party service listed on the caveat to obtain a discharge. The service performs the necessary checks, and if those succeed, it issues a discharge to Bob. Bob (recursively) performs the above procedure for any third-party caveats on the discharge, and once all discharges have been obtained, he presents all of them with the blessing B_{Bob} to the TV.
- The TV invokes $\mathsf{IsValidBlessing}(B_{Bob}, \mathcal{R}_{TV}, \mathcal{C}_{TV})$ to verify that the blessing B_{Bob} is valid for the current context from the TV's perpspective ($\mathcal{C}_{TV}$) and the set of blessing roots recognized by the TV ($\mathcal{R}_{TV}$). If this step succeeds, the TV verifies that the name of the blessing (`Alice / Houseguest / Bob`) satisfies its access control policy (`Allow Alice:Alice / Houseguest`). The connection is aborted if any of these checks fail.

– After authorization succeeds at the TV's end, the protocol is complete and an encrypted channel is established between Bob and the TV. Application data pertaining to the RPC is then exchanged on this channel.

4.8 Practical Considerations

We now discuss some considerations involved in deploying the authorization model in practice.

Managing Blessings. Authorization in Vanadium is based on blessings. A principal may acquire multiple blessings over time, each providing access to some set of services under some contextual restrictions. Consequently, managing these blessings may become quite onerous. The first problem is storing all these blessings while keeping track of the meta-data about where they were obtained from and under what constraints. Another problem is selecting which blessing to present when authenticating to a peer. While presenting all blessings and letting the peer choose the relevant one is convenient, it has the downside of leaking sensitive information, for e.g., a blessing may reveal that Bob is a house guest of Alice. Instead, Vanadium provides a means to selectively share blessings with appropriate peers. Blessing are stored by Vanadium principals using a mechanism similar to cookie jars in Web browsers. All blessings are stored with a blessing pattern identifying the peer to whom they should be presented. This pattern may be set based on information provided by the blessing granter. For example, Bob can add the blessing `Alice/Houseguest/Bob` with the peer pattern `Alice`. Thus, Bob will present this blessing only when communicating with services that have a blessing name matching this pattern. Any other service that Bob communicates with will not know that he has this blessing from Alice.

Blessing vs. Adding to an ACL. The careful reader may have noticed that Vanadium offers two mechanisms for granting access to a resource. For instance, consider Alice's TV with an ACL `Allow Alice` which means all principals with the blessing name `Alice` or an extension of it have access. Alice can grant access to her TV to another principal by either extending her `Alice` blessing to the other principal or by adding the other principal's blessing name to the ACL. The question then is how does one decide which method is appropriate in a given use-case. Granting a blessing is akin to handing out a capability, and thus in a way this question is that of deciding between granting a capability versus modifying an ACL. We recommend the following approach for making the choice.

The constraints on the access being delegated must be taken into consideration. If the access is meant to be long-lived and unconstrained then modifying the ACL is preferable as it allows the service administrator to audit and revoke access at any time. For instance, Alice may share access to her TV with her flatmate Dave by adding the pattern `Dave` to the TV's ACL. Later when Alice moves out, she can revoke Dave's access by removing this pattern. On the other hand, when the access is constrained then blessing with caveats is a more appropriate choice. For instance, Alice may delegate temporary access to her TV to her house guest Bob by blessing him under a short-lived time caveat.

The choice of how access is granted also affects the subsequent auditability of the delegated access. For instance, when Dave accesses Alice' TV he would use his own blessing (assuming Alice's TV trusts Dave as a blessing root) and his access would be recorded as `Dave`. However, when Bob accesses the TV he would use his blessing from Alice, and thus his access would be recorded as `Alice/Houseguest/Bob`. Finally, we note that the option to change an ACL may not always be available. When Bob wants to grant access to Alice's TV to his friend Carol, extending his blessing to Carol may be his only option as he may not have the authority to modify the TV's ACL.

Revocation. We now discuss mechanisms for revoking access. Revocation is easy when access is granted by adding to an ACL or a group, as it amounts to simply removing the added entry. It is more challenging when access is granted via blessing. One approach is to always constrain blessings with short-lived time caveats, thereby invalidating them automatically after a certain time. Principals would then have to periodically reach out to their blessing granters for a fresh blessing. This idea is similar to "reconfirmation" in SDSI [27]. Using a third-party caveat pointed at a revocation service offers a more systematic way of realizing this idea. The revocation service would issue a short-lived discharge for the caveat only if the blessing has not been revoked.

While such third-party caveats elegantly encode revocation restrictions, they suffer from the downside of requiring blessing holders to periodically connect to a revocation service. Specifically, they introduce a trade-off between how swiftly a blessing may be revoked, and how long things may operate when disconnected. This trade-off may be ameliorated to some extent if devices can recognize when they are offline and use different revocation timeouts in that case. Furthermore, in the home setting, a discharge service may be run on a WiFi access point, thereby requiring devices to maintain connectivity only to the local WiFi network.

5 Application: Physical Lock

In this section, we explain how the Vanadium authorization model may be applied to a physical lock. The application highlights the flexibility and decentralization aspects of the model. A network controlled lock is a common device found in many modern homes today. It allows a user to lock and unlock a door from their phone, and share access to it with visitors. Today, there are a number of manufacturers building locks for homes, garages, factory floors etc.

The authorization model for most existing products involves a global service service in the cloud, often controlled by the lock manufacturer, that is an authority on all credentials used to access the lock. Typically, the service must be accessed during setup and whenever access is delegated. As discussed in Sect. 2, this is undesirable as communicating with global services requires internet access, which may not be perfectly reliable. It can be quite frustrating for a user to be unable to share access to a lock due to lack of internet connectivity at the time sharing is initiated. Furthermore, compromising the manufacturer owned service

may allow attacker to unlock all locks managed by the manufacturer. In what follows, we present an authorization model for locks that is fully decentralized, and does not depend on access to an external service or identity provider.

Overview. The key idea is to have the lock be its own identity provider. When the lock is set up by its owner, it creates a self-signed blessing for itself, and extends this blessing to the owner. The blessing granted to the owner is effectively the *key* to the lock. All subsequent access to the lock is restricted to clients that can wield this blessing or extensions of it. Delegation of access is simply carried out by extending the *key* blessing.

5.1 Authorization Details

Consider a user Alice who just bought a brand new lock for the front door of her house. We walk through the steps of setting up identity and access control for the lock. We assume that Alice is interacting with the lock using another device, say her phone.

Claiming a New Lock. We assume that an out-of-box lock device comes with a pre-installed public and secret key pair, and a blessing from its manufacturer of the form `<manufacturer> / <serial no>`. The first step in setting up the lock is for Alice to name the lock and obtain a blessing for subsequently interacting with it. This is accomplished by invoking the `Claim` method on the lock that returns a blessing bound to the invoker's (Alice's) public key.

The invocation is through a Vanadium remote procedure call. The invoker authorizes the lock by verifying that it presents a blessing from the manufacturer with the expected serial number. An unconfigured lock authorizes any principal to invoke the `Claim` method on it, after which it considers the setup process complete and no longer allows invocation of that method.

After the `Claim` invocation is authorized, the lock creates a self-signed blessing with a name provided by the caller. This blessing is presented by the lock to authenticate to clients during all subsequent invocations. The lock then acts as an identity provider and extends this blessing to the invoker's public key (learned during authentication). This granted blessing is called the *key* blessing of the lock. The invoker saves this blessing for subsequent interactions with the lock and also recognizes its root as an identity provider. For instance, Alice may claim the lock on her front door with the name `AliceFrontDoor`. The lock will subsequently authenticate to others as `AliceFrontDoor`, and would grant the blessing `AliceFrontDoor / Key` to Alice (here `Key` is the blessing extension used by the lock).

Locking and Unlocking. Once a lock has been claimed, the `Claim` method is disabled and the lock instead exposes the `Lock` and `Unlock` methods. The methods are protected by an ACL that allows access only to clients that wield a blessing from the lock's identity provider. In the above example, the ACL would be `Allow  AliceFrontDoor`, which would be matched by the blessing `AliceFrontDoor / Key`.

Delegating Access. Any extension of the key blessing also matches the ACL for the lock's methods, and thus access to the lock can be delegated by extending this blessing. As usual, caveats can be added to the extension to restrict its scope. For instance, Alice can extend her blessing `AliceFrontDoor/Key` to her house cleaner as `AliceFrontDoor/Key/Cleaner` under a time caveat that is valid only on Mondays between 8AM to 10AM.

Auditing Access. The lock can keep track of the blessings used to access it, even ones that have invalid caveats. Thus, Alice can inspect the log on the lock for auditing access attempts made by her house cleaner. In particular, Alice can detect if the cleaner tried to access the lock outside of the agreed upon time (8AM to 10AM on Mondays) or if there is access by someone who has received a delegated blessing from the cleaner.

5.2 Discussion

We highlight three distinguishing aspects of the authorization model presented in this section.

Decentralized. Each lock is an authority on the secrets and credentials that can be used to access it. No external entity, including the lock manufacturer, can mint credentials to access a claimed lock device. The credentials for accessing one lock are completely independent from those for accessing another. Thus, attackers have no single point of attack to unlock multiple instances.

No Internet Connectivity Required. The authorization model does not require the lock or the device interacting with it to have internet access at any point, including during setup. The model does not rely on any third-party service or identity provider.

Audited. The lock can keep track of when it was accessed, by whom (represented by the blessings). Since blessings inherently capture a delegation trail, the access log also conveys how the invoker obtained access.

Having highlighted the above advantages, we note that decentralization comes at a cost. For instance, the lack of an authoritative source in the cloud makes it hard to recover from loss or theft of blessings and secret keys. Furthermore, users are responsible for managing multiple blessings and keeping track of various delegations they make. We believe that carefully designed user-interfaces and appropriate reset modes for the lock device can help address some of these concerns.

6 Conclusion

This tutorial presents the authorization model of the Vanadium framework [6]. In this model, each principal has a digital signature public and secret key pair, and a set of hierarchical human-readable names bound to its public key via

certificate chains called *blessings*. All authorizations associated with a principal are based on its blessing names. In particular, a principal makes a request to a service by presenting one of its blessings (using the Vanadium authentication protocol [7]), and the request is authorized if the blessing name satisfies the service's access control policy.

A notable feature of the model is its support for decentralization and fine-grained delegation. The model does not require the existence of special identity providers that are trusted by all principals by default. Instead, any principal may choose to become an identity provider and create blessings for itself and other principals. Each principal has a choice over which other principals it recognizes as identity providers. Only blessings from recognized identity providers are considered valid. In practice, we anticipate that there will be a small set of large-scale identity providers that most people will commonly use in interactions with the wider world.

Principals can delegate access by extending one or more of their blessings to other principals. The scope of delegations can be constrained very finely by adding caveats [10] to the delegated blessing. In particular, blessings support third-party caveats which allow predicating delegations on consent by specific third-parties. Such caveats elegantly support revocation, proximity-based restrictions, and audit requirements.

Access control policies in Vanadium may indirect through groups whose definition may be distributed across multiple servers, possibly under different administrative domains. While several access control mechanisms support groups, a distinguishing aspect of our design is using groups to construct compound names. For instance, the pattern $\texttt{AliceFriends}_{\texttt{G}}\,/\,\texttt{Phone}$, with $\texttt{AliceFriends}_{\texttt{G}}$ being a group of blessing names of Alice's friends, defines the set of blessing names of phones of Alice's friends. Such compound names coupled with negative clauses in ACLs make the problem of checking group membership fairly complex, especially when the group server may be unreachable. The Vanadium authorization model mitigates some of the difficulties by making simplifying choices. For instance, group definitions cannot contain negative clauses, and ACLs cannot be reused for defining groups or other ACLs. This tutorial provides a brief overview of our solution, with a more comprehensive description available in [8].

Future Directions. There are several future directions for this line of work. The first and perhaps the most important direction is on making the model and its primitives easily usable by end users. This is paramount to the adoption of the model. Mechanisms for making the model more usable may include designing intuitive user-interfaces for visualizing, granting and revoking blessings, and conventions on blessing names that help write intelligible access-control policies.

Another direction is that of enabling *mutual* privacy in the Vanadium authentication protocol. Currently, the protocol involves the server presenting its blessing before the client. While this is beneficial to the client, as it may choose to not reveal its blessing after seeing the server's blessing, it is disadvantageous to the server. The server's blessing is effectively revealed to anyone who connects to it, including active network attackers. In fact, this lack of mutual privacy

is common to many other mutual authentication protocols (such as TLS [17], SIGMA-I [15]) wherein one of the parties must reveal its identity first. In the scenarios considered in this work, the participants may be personal end-user devices neither of which is inclined to reveal its identity before learning the identity of its peer. In ongoing work [31], we are designing a private mutual authentication protocol that allows each end to learn its peer's blessing only if it satisfies the peer's authorization policy.

Finally, one may consider designing mechanisms for securely leveraging external cloud-based services when internet access is available. For instance, a Cloud-based service may be used as a transparent proxy for RPCs, as a revocation and auditing service for blessing delegations, or as a readonly backup for data. In all cases, the goal would be to leverage external Cloud-based services for various tasks while granting them the minimal authority necessary for the task.

Acknowledgments. This work is a result of a joint effort by several members of the Vanadium team at Google. We would like to thank Martín Abadi, Mike Burrows, Ryan Brown, Bogdan Caprita, Thai Duong, Cosmos Nicolaou, Himabindu Pucha, David Presotto, Adam Sadovsky, Suharsh Sivakumar, Gautham Thambidorai, Robin Thellend for their contributions to designing and implementing the Vanadium authorization model. We are grateful to Martín Abadi and Mike Burrows for helpful comments on drafts of this tutorial.

References

1. Fridge sends spam emails as attack hits smart gadgets. http://www.bbc.com/news/technology-25780908
2. Hackers remotely kill a jeep on the highway? with me in it. https://www.wired.com/2015/07/hackers-remotely-kill-jeep-highway/
3. Openid. http://openid.net/
4. Smart meters can be hacked to cut power bills. http://www.bbc.com/news/technology-29643276
5. The Internet of Things is wildly insecure? and often unpatchable. https://www.schneier.com/essays/archives/2014/01/the_internet_of_thin.html
6. Vanadium. http://vanadium.github.io/
7. Vanadium Authentication Protocol. https://vanadium.github.io/designdocs/authentication.html
8. Abadi, M., Burrows, M., Pucha, H., Sadovsky, A., Shankar, A., Taly, A.: Distributed authorization with distributed grammars. In: Bodei, C., Ferrari, G.-L., Priami, C. (eds.) Programming Languages with Applications to Biology and Security. LNCS, vol. 9465, pp. 10–26. Springer, Heidelberg (2015)
9. Appel, A., Felten, E.: Proof-carrying authentication. In: CCS, pp. 52–62 (1999)
10. Birgisson, A., Politz, J.G., Erlingsson, U., Taly, A., Vrable, M., Lentczner, M.: Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In: NDSS (2014)
11. Blaze, M., Feigenbaum, J., Ioannidis, J.: The KeyNote Trust-Management System Version 2. RFC 2704 (Proposed Standard), September 1999
12. Blaze, M., Feigenbaum, J., Lacy, J.: Decentralized trust management. In: IEEE Symposium on Security and Privacy, pp. 164–173 (1996)

13. Borisov, N., Brewer, E.: Active certificates: a framework for delegation. In: NDSS, pp. 30–40 (2002)
14. Braz, C., Robert, J.: Security and usability: the case of the user authentication methods. In: Conference on L'Interaction Homme-Machine, pp. 199–203 (2006)
15. Canetti, R., Krawczyk, H.: Security analysis of IKE's signature-based key-exchange protocol. In: Yung, M. (ed.) CRYPTO 2002. LNCS, vol. 2442, pp. 143–161. Springer, Heidelberg (2002)
16. Clarke, D., Elien, J., Ellison, C., Fredette, M., Morcos, A., Rivest, R.: Certificate chain discovery in SPKI/SDSI. J. Comput. Secur. **9**, 285–322 (2001)
17. Dierks, T., Rescorla, E.: The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008
18. Ellison, C., Frantz, B., Lampson, B., Rivest, R., Thomas, B., Ylonen, T.: SPKI Certificate Theory. RFC 2693 (Proposed Standard), September 1999
19. Hewlett Packard Enterprise: Internet of things research study. http://www8.hp.com/h20195/V2/GetPDF.aspx/4AA5-4759ENW.pdf
20. Gong, L.: A secure identity-based capability system. In: IEEE Symposium on Security and Privacy, pp. 56–63 (1989)
21. Hardt, E.: The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard), October 2012
22. Lampson, B., Abadi, M., Burrows, M., Wobber, E.: Authentication in distributed systems: theory and practice. In: SOSP, pp. 165–182 (1991)
23. Li, N., Feigenbaum, J., Grosof, B.N.: A logic-based knowledge representation for authorization with delegation. In: CSFW, pp. 162–174 (1999)
24. Myers, M., Ankney, R., Malpani, A., Galperin, S., Adams, C.: X.509 Internet Public Key Infrastructure Online Certificate Status Protocol - OCSP. RFC 2560 (Proposed Standard), June 1999
25. Neuman, B.C.: Proxy-based authorization and accounting for distributed systems. In: ICDCS, pp. 283–291 (1993)
26. Rapid7. Hacking IoT: A case study on baby monitor exposures and vulnerabilities. https://www.rapid7.com/resources/iot/baby-monitors.jsp
27. Rivest, R.L., Lampson, B.: SDSI - a simple distributed security infrastructure. Technical report (1996). http://people.csail.mit.edu/rivest/sdsi11
28. Santesson, S., Farrell, S., Boeyen, S., Housley, R., Polk, W.: Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280 (Proposed Standard), May 2008
29. Schneider, F.B.: Untitled textbook on cybersecurity. Chap. 9: Credentials-based authorization (2013). http://www.cs.cornell.edu/fbs/publications/chptr.CredsBased.pdf
30. Whitten, A., Tygar, J.D.: Why Johnny can't encrypt: a usability evaluation of PGP 5.0. In: USENIX Security Symposium, pp. 169–183 (1999)
31. Wu, D.J., Taly, A., Shankar, A., Boneh, D.: Privacy, discovery, and authentication for the internet of things (2016). https://arxiv.org/abs/1604.06959
32. Zimmermann, P.R.: The Official PGP User's Guide. MIT Press, Cambridge (1995)