Distributed Authorization with Distributed Grammars

Martín Abadi, Mike Burrows, Himabindu Pucha,
Adam Sadovsky, Asim Shankar, and Ankur Taly

Google, Mountain View, California, USA

Abstract. While groups are generally helpful for the definition of authorization policies, their use in distributed systems is not straightforward. This paper describes a design for authorization in distributed systems that treats groups as formal languages. The design supports forms of delegation and negative clauses in authorization policies. It also considers the wish for privacy and efficiency in group-membership checks, and the possibility that group definitions may not all be available and may contain cycles.

1 Introduction

Groups provide a useful level of indirection for authorization policies, in particular those described by access control lists (ACLs). When ACLs refer to groups, the ACLs can be simple and short. For example, an ACL may permit access to all principals in the group `FriendlyClients`, which itself consists of users in the group `Friends` with devices in the group `Devices` via programs in the group `TrustedApps`. The definitions of these groups can be managed separately from the ACL, and shared by many other ACLs (e.g., [6]).

In distributed systems, the use of groups is not straightforward (e.g., [4, 5]). First, it requires a distributed scheme for naming groups. Even with such a scheme, group definitions may not all be available at the time of an ACL check; they may have unintended consequences or circularities that no single participant in the system can detect locally; and the entities that control them may not all be equally trusted. In addition, lookups of group membership may incur the costs of remote communication; and, in general, there is no guarantee of atomicity of lookups across groups. Finally, the lookups need to be secure and provide appropriate privacy guarantees. No universal solution to these difficulties seems likely to emerge because each system faces different trade-offs and constraints.

This paper describes a design for access control with groups in a new set of libraries, tools, and services that aim to simplify the process of building distributed applications. Our design supports forms of delegation, via local names. It also supports negative clauses in ACLs, with a conservative semantics when group definitions are not available or contain cycles. Moreover, it addresses the wish for privacy and efficiency in group-membership checks—at least in the sense that the dissemination of group memberships occurs in response to relevant queries, not promiscuously.

In this setting, each principal is identified with a public key, but it typically has one or more human-readable names, which we call blessings.[1] Concretely, these blessings are granted in public-key certificate chains bound to the principal's public key. For example, a television set owned by a principal with the blessing `Alice` may have the blessing `Alice / TV`. Here, `TV` is a local name, which, much as in previous systems and languages for security (e.g., [7, 2]), any principal can generate and apply autonomously. Principals may have multiple blessings, each reflecting the principal that granted it. For example, the same television set may also have the blessing `SomeCorp / TV123` from its manufacturer.

Blessings are the basis for authentication and authorization. Specifically, the "`Bless`" operation allows a principal to extend one of its blessings and create a blessing bound to another principal's public key, thereby delegating the authority associated with the blessing. For example, an ACL may include the clause `Allow Alice / TV`, so that all principals with a blessing that matches `Alice / TV` will have access to the object that the ACL protects, and a principal with the blessing `Alice` may choose to grant the blessing `Alice / TV` to its television set. In practice, the delegation of authority is seldom unconditional. Caveats [1] can restrict the conditions under which blessings are usable, for instance limiting their validity to a certain time period; we do not discuss these caveats further in this paper, since their generation and validation precedes the access-control checks on which we focus.

Our design supports groups that contain not only atomic names such as `Alice` and `TV`, but also longer, compound blessings such as `Alice / TV`. Furthermore, the definition of a group may refer to other groups at the top level (e.g., "`Friends` includes `OldFriends`") and as part of compound blessings (e.g., "`FriendlyClients` includes `Friends / Devices / TrustedApps`"). An important theme of the design is to regard groups as formal languages, with group definitions inducing grammar productions. Unlike in traditional formal languages, however, the grammar productions are distributed, so we have to consider concerns such as communication costs, availability, and privacy. The analogy is helpful despite these differences.

By now, many other systems support distributed authorization, in various ways. On the other hand, the combination of local names, groups, and negative clauses is, to our knowledge, rather uncommon and subject to limitations. Beyond the immediate value of our work, we hope that it contributes to shedding light on some of the difficulties and options for systems with these features.

The next section introduces the definitions of blessings, groups, ACLs, and related concepts. Section 3 gives a semantics to blessing patterns. Section 4 provides a simple but impractical definition of the semantics of ACLs, as a spec-

[1] Strictly speaking, the term blessing refers to a certificate chain, and the term blessing name refers to the human-readable name specified in the certificate chain. Blessing name is often abbreviated to blessing when there is no risk of confusion, as in the present paper. Below, we use the term blessing rather broadly: we consider that `/`-separated sequences of names n_1 `/` ... `/` n_k are blessings even when they might never be related to public keys.

ification. Section 5 then outlines a distributed implementation of this semantics. Section 6 elaborates on the rationale for one delicate aspect of the semantics of ACLs. Section 7 concludes.

2 Basics: Blessings, Groups, and ACLs

In this section, we define blessings, ACLs, and also blessing patterns, which are generalizations of blessings that allow references to groups.

2.1 Ordinary Names and Group Names

We assume a set of group names, and a disjoint set of other names that we call ordinary names. We let g range over group names, and n over ordinary names.

In our implementation, ordinary names and group names have quite different forms and usages. In particular, each group name suffices for determining an appropriate server who can answer questions about the group and for querying that server. On the other hand, ordinary names are fundamentally local names, simple strings that can be interpreted differently across a system. They may refer to a variety of entities (users, services, programs, program versions, ...). They may however be subject to conventions.

2.2 Blessings and Blessing Patterns

The syntax of blessings and blessing patterns is given by the following grammar:

$$
\begin{array}{lll}
B ::= n & & \text{blessings} \\
\quad \mid \; n \texttt{ / } B & & \\
 & & \\
P ::= n & & \text{blessing patterns} \\
\quad \mid \; g & & \\
\quad \mid \; n \texttt{ / } P & & \\
\quad \mid \; g \texttt{ / } P & &
\end{array}
$$

Here, B ranges over blessings and P over blessing patterns; `/` is a binary operator for forming blessings and blessing patterns. Thus, a blessing is a non-empty sequence of ordinary names, separated by `/`, while a blessing pattern is a non-empty sequence of ordinary names and group names, separated by `/`. We take `/` to be associative.

For example, if `Alice` and `Phone` are ordinary names and `Friends` and `Devices` are group names, then:

- `Alice` and `Alice / Phone` are blessings, and they are also blessing patterns;
- so are `Alice / Alice`, `Phone / Phone`, and `Phone / Alice`, though they are not necessarily meaningful—we do not have a type system or other constraints that would prevent such expressions;

– `Friends`, `Friends / Phone`, `Alice / Devices`, and `Friends / Devices` are all blessing patterns, but not blessings.

We write *AllBlessings* for the set of all blessings. When B and B' are blessings, we write $B \preceq B'$, and say that B is a prefix of B', if the sequence of names in B is a prefix of that in B'. We take this prefix relation to be reflexive, not strict; that is, every blessing is a prefix of itself.

We often have to manipulate lists of blessings and lists of blessing patterns. In particular, below, lists of blessings are an input to ACL checks; lists of blessing patterns appear in group definitions. Therefore, we introduce syntactic categories for them:

$$\begin{array}{rcll} M & ::= & \texttt{empty} & \text{lists of blessings} \\ & | & B,\ M & \\ \\ L & ::= & \texttt{empty} & \text{lists of blessing patterns} \\ & | & L,\ P & \end{array}$$

We use the constant `empty` to represent the empty list, and use comma as a binary operator for forming lists. We often omit `empty`, and for example may write the list `empty`, `Alice`, `Bob` as `Alice`, `Bob`.

2.3 Groups

Group names are of two sorts: those for built-in groups and those for defined groups. In both cases, a group can be thought of as a set of blessings.

Built-in Groups Some groups are provided by the underlying platform, so do not require extensional definition. The set of all blessings, to which we refer by the name `AllBlessings`, is an example. Another example—of much narrower interest—might be the set of blessings of the form n_1 `/` n_2 such that n_1 identifies a sports team in a particular league and n_2 identifies one of the players in n_1's roster. We write *BuiltInGroups* for the set of names of these built-in groups.

In general, built-in groups may be implemented by fairly arbitrary pieces of code that answer, in particular, membership queries. Below we discuss the interface that such code should provide.

Formally, we assume a function *Elts* that maps each $g \in \mathit{BuiltInGroups}$ to a set of blessings (intuitively, the elements of g). In this paper, for simplicity, the function *Elts* is fixed—in particular, independent of who computes it and of the definitions of defined groups. For instance, we let $\mathit{Elts}(\texttt{AllBlessings}) = \mathit{AllBlessings}$. As in this case, a set $\mathit{Elts}(g)$ may be infinite.

Defining Groups Other group names may be associated with definitions that equate a group name with a list of blessing patterns:

$$g =_{\text{def}} L$$

Given a set *DefSet* of definitions $\{g_1 =_{\text{def}} L_1, \ldots, g_k =_{\text{def}} L_k\}$, we require that the group names g_i be pairwise distinct and distinct from elements in *BuiltInGroups*. As long as each group name is associated with a server, this requirement is easy to enforce in a distributed manner.

On the other hand, we do not require the absence of cycles in the definitions, primarily because we do not count on being able to enforce this requirement in a distributed manner. Secondarily, some simple cycles may occasionally be useful. For example, the definitions

$$\texttt{Gadgets} =_{\text{def}} \texttt{TV}, \texttt{Devices}$$
$$\texttt{Devices} =_{\text{def}} \texttt{Phone}, \texttt{Gadgets}$$

have the effect of equating `Devices` with `Gadgets` while allowing two different servers to include `TV` and `Phone` in this group. As another example, the definition

$$\texttt{DeviceChains} =_{\text{def}} \texttt{Devices}, \texttt{Devices / DeviceChains}$$

lets `DeviceChains` consist of blessings formed by sequences of elements of the group `Devices`. So, we may warn about cycles, and we may discourage their use, but we aspire to provide a clean, helpful semantics at least for simple cycles, and a conservative semantics for all cycles.

We allow the possibility that some group names are neither in *BuiltInGroups* nor have a definition (at least not an available definition). We aim to provide a conservative semantics for those names.

Time Both the code associated with built-in groups and the definitions associated with other group names may change over time. They may even change during an ACL check. Correctness expectations may have to be relaxed accordingly (for example, so as to allow reordering queries to servers.) Although the definitions and algorithms presented in this paper are mostly silent on this matter, we discuss it briefly in Section 5.4.

2.4 ACLs

An ACL is a list of clauses, each of which permits or denies access to principals that present blessings that match a particular blessing pattern:

$$\begin{array}{lll} A ::= \texttt{empty} & & \text{ACLs} \\ \quad | \;\; A, \texttt{Allow}\ P & & \\ \quad | \;\; A, \texttt{Deny}\ P & & \end{array}$$

Our present implementation requires that all `Allow` clauses precede all `Deny` clauses, but this paper treats a more general syntax with arbitrary alternations.

Our semantics of ACLs is order-dependent. Basically, later ACL entries will win over earlier ones according to the specification of Section 4. For example, when `Alice` is in the group `Friends`, the ACL `Deny Alice, Allow Friends` will permit access with the blessing `Alice` but the ACL `Allow Friends, Deny Alice`

will deny it. The default is to deny access, so for example neither the ACL `empty` nor the ACL `Allow Alice` will permit access with the blessing `Bob`. The specification of Section 4 also addresses other aspects of the semantics of ACLs, and in particular the rules for matching blessings against the blessing patterns in clauses, which rely on the prefix relation $\preceq$.

We abbreviate ACLs by combining consecutive `Allow` clauses, for example writing `Allow Friends, Alice` for `Allow Friends`, `Allow Alice`, and similarly for consecutive `Deny` clauses.

We expect that many ACLs will be of the simple form `Allow` g, where g is a group name. More generally, many may be of the form `Allow` $P_1, \ldots, P_k$, or perhaps

$$\texttt{Allow } P_1, \ldots, P_k, \texttt{ Deny } P_{k+1}, \ldots, P_{k+k'}$$

where $P_1, \ldots, P_{k+k'}$ are blessing patterns. On the other hand, ACLs with many alternations of `Allow` and `Deny` clauses

$$\texttt{Allow } P_1, \texttt{ Deny } P_2, \ \ldots, \ \texttt{Allow } P_{k+k'-1}, \texttt{ Deny } P_{k+k'}$$

should arise only in advanced cases, as they can be hard to understand.

The current syntax does not allow naming ACLs. This limitation means that sharing happens through named groups.

3 Semantics

Intuitively, each blessing pattern—and, in particular, each group name—denotes a set of blessings. This section defines how we map blessing patterns to sets of blessings.

3.1 The Meaning of Blessing Patterns

Assuming a semantics of group names (a mapping from group names to sets of blessings, given as a parameter ρ), the function `Meaning` maps blessing patterns and lists of blessing patterns to sets of blessings. It is defined inductively as follows, first for blessing patterns:

$$\begin{aligned}
\texttt{Meaning}_\rho(n) &= \{n\} \\
\texttt{Meaning}_\rho(g) &= \rho(g) \\
\texttt{Meaning}_\rho(n \texttt{ / } P) &= \{n \texttt{ / } s \mid s \in \texttt{Meaning}_\rho(P)\} \\
\texttt{Meaning}_\rho(g \texttt{ / } P) &= \{s \texttt{ / } s' \mid s \in \texttt{Meaning}_\rho(g), s' \in \texttt{Meaning}_\rho(P)\}
\end{aligned}$$

and then for lists of blessing patterns:

$$\begin{aligned}
\texttt{Meaning}_\rho(\texttt{empty}) &= \emptyset \\
\texttt{Meaning}_\rho(L, \ P) &= \texttt{Meaning}_\rho(L) \cup \texttt{Meaning}_\rho(P)
\end{aligned}$$

3.2 From Group Definitions to Grammars and Languages

A semantics of group names is basically a function ρ that maps each group name to the set of members of the group. However, we have to decide what happens when an expression (for instance, an ACL) refers, directly or indirectly, to a group that has not been defined. In a distributed setting (when the definitions are at different servers), we also have to decide what happens when the group definition may exist but cannot be looked up, for whatever reason. Since we wish to be conservative (fail-safe), our decision may be different in `Allow` and `Deny` clauses. For this reason, we define not one function but two functions, called $\rho_{\Downarrow}$ and $\rho_{\Uparrow}$. They coincide in the case in which all references to groups can be resolved.

For the construction of $\rho_{\Downarrow}$, we regard a list of group definitions *DefSet* as inducing formal languages, as follows.

- Ordinary names and `/` are terminals.
- Group names are non-terminals.
- We associate productions with the group definitions, for example turning a definition

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

  into the two productions

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

- We also associate productions with each built-in-group name g: $g \to B$ for each $B \in \mathit{Elts}(g)$.
- Finally, we do not associate productions with any remaining group names (those that are neither defined in *DefSet* nor built-in-group names).

For each group name g, we let $\rho_{\Downarrow}(g)$ be the set of blessings generated from g by these productions. Thus, the question of group membership can be reduced to that of formal-language membership.

When $\mathit{Elts}(g)$ is finite for each $g \in \mathit{BuiltInGroups}$, the productions above constitute a context-free grammar. Otherwise, we still obtain a formal language $\rho_{\Downarrow}(g)$ for each group name g, though these need not be context-free languages. More precisely, much as in formal-language theory, $\rho_{\Downarrow}$ is the least fixed-point of the function F such that, for every g,

- $F(\rho)(g) = \texttt{Meaning}_{\rho}(L)$ if g is defined by $g =_{\text{def}} L$;
- $F(\rho)(g) = \mathit{Elts}(g)$ for $g \in \mathit{BuiltInGroups}$; and
- $F(\rho)(g) = \emptyset$ otherwise.

The existence of this least fixed-point follows from the facts that $\texttt{Meaning}_{\rho}$ is monotone as a function of ρ and that *Elts* does not depend on ρ.

The construction of $\rho_{\Uparrow}$ is analogous, except that in the last case we let $F(\rho)(g) = \mathit{AllBlessings}$. In particular, we still take a least fixed-point (not a greatest fixed-point).

In practice, we need not always compute least fixed-points: we may allow ourselves to treat some group definitions as being unavailable whenever we wish—for instance, when the corresponding server has failed, or when we have exhausted our computational budget. The result will be a conservative approximation. Section 5 follows this approach.

Regular expressions are fairly common in access control, for example in defining firewall rules. The Singularity security model used them for defining groups, without negation, via a file-system name-space [8]. With the definitions above, we go beyond regular languages, not because we expect to need the full power of context-free languages (and perhaps more), but in order to avoid cumbersome syntactic conditions in group definitions. Still, it is conceivable that restricting attention to regular languages would have advantages.

4 Specifying Authorization Checks

A principal that has collected multiple blessings may present a subset for the purposes of an authorization decision. It may decide not to present all its blessings, perhaps because of concerns about performance or confidentiality. However, it should not gain additional rights by virtue of withholding some blessings.

Accordingly, the function that performs authorization checks, `IsAuthorized`, is applied to a list of blessings M and an ACL A. It decides whether access should be granted according to A when the blessings in M are presented. It is defined in terms of an auxiliary function $\texttt{IsAuthorized}_1(B, A)$ that is applied to a single blessing B and an ACL A. This auxiliary function works by cases on the three possible forms of A, namely (1) `empty`, (2) A', `Allow` P for some A' and P, and (3) A', `Deny` P for some A' and P. In the first case, it returns `false`; in the second and the third, it checks B against P and against A', then returns an appropriate boolean combination of the results. If B happens to match both `Allow` and `Deny` clauses in A, later clauses win over earlier ones. Since each blessing B is treated separately, $\texttt{IsAuthorized}(\cdot, A)$ is monotonic in its first argument, as desired.

$$\texttt{IsAuthorized}(M, A) = \exists B \in M.\texttt{IsAuthorized}_1(B, A)$$

$$\begin{array}{l}\texttt{IsAuthorized}_1(B, A) = \\ \quad \texttt{case } A \texttt{ of} \\ \quad\ \ \texttt{empty} : \texttt{false} \\ \quad \mid\ A', \texttt{Allow } P : (\exists B' \in \texttt{Meaning}_{\rho_\Downarrow}(P).B' \preceq B) \vee \texttt{IsAuthorized}_1(B, A') \\ \quad \mid\ A', \texttt{Deny } P : (\neg\exists B' \in \texttt{Meaning}_{\rho_\Uparrow}(P).B' \preceq B) \wedge \texttt{IsAuthorized}_1(B, A')\end{array}$$

This definition relies on the mappings $\rho_\Downarrow$, $\rho_\Uparrow$, and `Meaning`, described in Section 3. It is intended as a specification, without a directly evident concrete implementation.

ACL clauses that refer (directly or indirectly) to undefined groups are treated conservatively by relying on $\rho_\Downarrow$ and $\rho_\Uparrow$ depending on the type of clause. This conservative treatment is done "one entry at a time". In some cases, this approach

might yield slightly surprising (but safe) results. Let us consider, for example, the unusual ACL

`Allow Alice`, `Deny Friends`, `Allow Friends`

where `Friends` is a group name but it has no corresponding definition, or its definition is unavailable, and `Friends` is not in *BuiltInGroups*. Suppose that we wish to know whether this ACL allows access to a request with the blessing `Alice`. We start from the end. The clause `Allow Friends` does not permit access, because we make the conservative assumption that `Friends` is empty. The clause `Deny Friends` denies access, because we make the conservative assumption that `Friends` contains all blessings. So we never look at `Allow Alice`, and deny access! Although this outcome may be counterintuitive, it is conservative, and seems adequate because we do not expect to give pleasing results when groups are undefined or their definitions are unavailable. One may certainly imagine more elaborate approaches, perhaps with some form of symbolic constraint-solving.

The definition uses the prefix relation $\preceq$ (instead of requiring exact equality) for checking both `Allow` and `Deny` clauses. While this consistency is certainly attractive, the choice of $\preceq$ has different significance in the two cases:

- For `Allow` clauses, the use of the relation $\preceq$ is a matter of convenience. For example, when one writes the ACL `Allow Alice` for an object that one wishes to share with a principal with the blessing `Alice`, it is typically expedient that this principal gain access even when this access may happen via a phone with the blessing `Alice / Phone`. Thus, lengthening a blessing does not reduce authority with respect to the `Allow` clauses in ACL checks. The longer blessing is however not equivalent to the shorter one in other respects: the longer blessing may trigger a `Deny` clause, and a principal that holds the blessing `Alice / Phone` cannot in general obtain other extensions of `Alice`, such as `Alice / TV`.
  This semantics is definable from a semantics that requires exact equality. For example, under the latter semantics, one could write the ACL

  `Allow Alice`, `Alice / AllBlessings`

  rather than `Allow Alice`. Conversely, even with the semantics that uses $\preceq$ it is possible to define ACLs that insist on exact equality. For example, one can write

  `Allow Alice`, `Deny Alice / AllBlessings`

  Alternatively,[2] assuming that `eob` is a reserved name that appears in blessings only at the end, one can write

  `Allow Alice / eob`

[2] In general, these two approaches do not always yield equivalent results. Suppose that the group g is defined to contain `Alice` and `Alice / Phone`. The ACL `Allow` g, `Deny` g `/ AllBlessings` denies access with `Alice / Phone`, while the ACL `Allow` g `/ eob` allows access with `Alice / Phone / eob`. Both ACLs deny access with `Alice / Phone / FunnyApp` and `Alice / Phone / FunnyApp / eob`.

– For `Deny` clauses, it generally does not make sense to forbid access with the blessing B but to permit it with a longer blessing, from a security perspective. Whoever has B would be able to extend it in order to circumvent the `Deny` check.

One may be tempted by an even weaker criterion for matching in `Allow` clauses, which we call "prefix matching". For example, with this criterion, the ACL `Allow Alice / Phone` would permit access with the blessing `Alice`. The main motivation for this decision is that denying this access has no clear security benefit: whoever holds `Alice` could form `Alice / Phone` in order to gain access. Section 6 discusses prefix matching in more detail and explains why we have not adopted it.

5 An Implementation of Authorization Checks

The function `IsAuthorized`, as defined above, might be implemented by calculating the functions $\rho_{\Downarrow}$, $\rho_{\Uparrow}$, and `Meaning` at the relying party, then applying the definitions blindly. However, these calculations generally require knowledge of the group definitions, which we may not want to disseminate for reasons of efficiency and privacy. The relevant groups might even be infinite, so we cannot enumerate them in general. Moreover, a full computation of `Meaning` is sometimes not required for determining if some particular blessing is or is not a member of the corresponding set of blessings. Therefore, we consider distributed, query-driven implementations of `IsAuthorized`. We first reduce `IsAuthorized` to a basic function `R`, then we discuss how to implement the required invocations of `R`. Basically, we rely on a form of top-down parsing.

Other algorithmic approaches may perhaps be derived from work on formal languages. More speculatively, the connection with formal languages suggests problems in secure multiparty computation (e.g., [3]): if several parties hold parts of a context-free grammar, can they cooperate to establish membership of a string in the corresponding language while not revealing any other information? General results on secure multiparty computation indicate that they can, but an efficient solution does not seem straightforward.

5.1 An Auxiliary Function: `R`

Suppose that we wish to know whether a particular blessing is in $\texttt{Meaning}_{\rho}(P)$. When the blessing is an ordinary name n, we may proceed as follows:

– If $P = m$ or $P = m \texttt{ / } P_1$ or $P = n \texttt{ / } P_1$ or $P = g \texttt{ / } P_1$ then fail, for every $m \neq n$ and every group name g.
– If $P = n$ then succeed.
– If $P = g$ then ask a server responsible for g whether n is an element of g and return the result.

When the blessing is a compound blessing $n \texttt{ / } B_1$, we may instead proceed as follows:

- If $P = m$ or $P = m \mathbin{/} P_1$ then fail, for every $m \neq n$.
- If $P = n$ then fail.
- If $P = n \mathbin{/} P_1$ then recurse with B_1 and P_1.
- If $P = g$ then ask a server responsible for g whether $n \mathbin{/} B_1$ is an element of g and return the result.
- If $P = g \mathbin{/} P_1$ then ask a server responsible for g whether there exist B_2, B_3 such that $n \mathbin{/} B_1 = B_2 \mathbin{/} B_3$, and B_2 is an element of g, and if so recurse with B_3 and P_1 (for each suitable B_3, for completeness).

Thus, a server responsible for g needs to answer questions of the following forms:

- whether a blessing B is in $\texttt{Meaning}_\rho(g)$,
- whether a blessing B can be written in the form $B_2 \mathbin{/} B_3$ where B_2 is an element of $\texttt{Meaning}_\rho(g)$.

While each question of the latter kind can be reduced to several questions of the former kind (one per prefix B_2 of B), providing an interface for asking questions of the latter kind allows a more direct, efficient interaction.

Therefore, we assume a function `R` with the following specification: `R` is such that, given a blessing B and a set of blessings S, $\texttt{R}(B, S)$ returns the set that consists of

- ϵ if $B \in S$, and
- every blessing B'' such that for some B' we have $B = B' \mathbin{/} B''$ and $B' \in S$.

Note that $\texttt{R}(B, S)$ may, in general, contain both blessings and ϵ. For example, if $S = \{n_1, n_1 \mathbin{/} n_2, n_1 \mathbin{/} n_2 \mathbin{/} n_3\}$ then $\texttt{R}(n_1 \mathbin{/} n_2, S) = \{\epsilon, n_3\}$. The name `R` stands for "rest", "remainder", or "residue".

Below we consider how to implement `R`.

5.2 Reducing `IsAuthorized` to `R`

Using `R`, we can reformulate the definition of `IsAuthorized`:

$$\texttt{IsAuthorized}(M, A) = \exists B \in M.\texttt{IsAuthorized}_1(B, A)$$

$$\begin{array}{l}\texttt{IsAuthorized}_1(B, A) = \\ \quad \texttt{case } A \texttt{ of} \\ \quad\ \texttt{empty} : \texttt{false} \\ \quad \mid\ A',\ \texttt{Allow } P : \texttt{R}(B, \texttt{Meaning}_{\rho_{\Downarrow}}(P)) \neq \emptyset \vee \texttt{IsAuthorized}_1(B, A') \\ \quad \mid\ A',\ \texttt{Deny } P : \texttt{R}(B, \texttt{Meaning}_{\rho_{\Uparrow}}(P)) = \emptyset \wedge \texttt{IsAuthorized}_1(B, A')\end{array}$$

This formulation is equivalent to our original one, but closer to our implementation.

5.3 Implementing the Calls to R

Next we consider how to compute and how to approximate $\mathtt{R}(B, \mathtt{Meaning}_{\rho_\Downarrow}(P))$ and $\mathtt{R}(B, \mathtt{Meaning}_{\rho_\Uparrow}(P))$ without fully expanding the definitions of $\rho_\Downarrow$, $\rho_\Uparrow$, and `Meaning`. We present basic algorithms first, then elaborate on distributed implementations.

We assume that we have $\mathtt{R}(B, \mathit{Elts}(g))$ for each $g \in \mathit{BuiltInGroups}$. In practice, this assumption means that the code that implements a built-in group g should offer an interface for asking queries of the form $\mathtt{R}(B, \mathit{Elts}(g))$. Note that $\mathtt{R}(B, \mathit{Elts}(g))$ is always finite, even when $\mathit{Elts}(g)$ is infinite. In the case of `AllBlessings`, this set consists of ϵ and the proper suffixes of B. We write $\mathtt{S}(B)$ for this set.

Basic Algorithms Suppose that we want functions $\mathtt{R}_\Downarrow$ and $\mathtt{R}_\Uparrow$ such that:

$$\mathtt{R}_\Downarrow(B, P) = \mathtt{R}(B, \mathtt{Meaning}_{\rho_\Downarrow}(P))$$
$$\mathtt{R}_\Uparrow(B, P) = \mathtt{R}(B, \mathtt{Meaning}_{\rho_\Uparrow}(P))$$

where $\mathtt{R}_\Downarrow$ and $\mathtt{R}_\Uparrow$ have, as implicit parameter, the group definitions *DefSet*. For brevity, we write $\mathtt{R}_X$ when we wish to refer to both $\mathtt{R}_\Downarrow$ and $\mathtt{R}_\Uparrow$ (but, in an equation such as $\mathtt{R}_X(\ldots) = \ldots \mathtt{R}_X(\ldots) \ldots$ we mean the same $\mathtt{R}_X$ on both sides). Given a list of blessing patterns $L = P_1, \ldots, P_k$, we let $\mathtt{R}_X(B, L) = \cup_{i=1..k} \mathtt{R}_X(B, P_i)$.

The desired functions $\mathtt{R}_\Downarrow$ and $\mathtt{R}_\Uparrow$ satisfy the equations:

$$\begin{array}{ll}
\mathtt{R}_X(n, n) & = \{\epsilon\} \\
\mathtt{R}_X(n, m) & = \emptyset \quad \text{if } m \neq n \\
\mathtt{R}_X(n \mathbin{/} B, n) & = \{B\} \\
\mathtt{R}_X(n \mathbin{/} B, m) & = \emptyset \quad \text{if } m \neq n \\
\mathtt{R}_X(n, m \mathbin{/} P) & = \emptyset \\
\mathtt{R}_X(n \mathbin{/} B, n \mathbin{/} P) & = \mathtt{R}_X(B, P) \\
\mathtt{R}_X(n \mathbin{/} B, m \mathbin{/} P) & = \emptyset \quad \text{if } m \neq n \\
\mathtt{R}_X(B, g) & = \begin{cases} \mathtt{R}_X(B, L) \text{ if } g =_{\text{def}} L \in \mathit{DefSet}, \text{ or else} \\ \mathtt{R}(B, \mathit{Elts}(g)) \text{ if } g \in \mathit{BuiltInGroups}, \text{ or else} \\ \emptyset \text{ if } X \text{ is } \Downarrow, \text{ or else} \\ \mathtt{S}(B) \text{ if } X \text{ is } \Uparrow \end{cases} \\
\mathtt{R}_X(B, g \mathbin{/} P) & = \{s \mid \exists s' \neq \epsilon . s' \in \mathtt{R}_X(B, g), s \in \mathtt{R}_X(s', P)\}
\end{array}$$

When oriented from left to right, these equations immediately suggest an algorithm for computing $\mathtt{R}_X(B, P)$. This algorithm proceeds by cases on the form of P. When P is not a group name and does not start with a group name, the algorithm then proceeds by cases on the form of B. When P is a group name g with a definition $g =_{\text{def}} L$ in *DefSet*, the algorithm unfolds this definition. When P is a group name $g \in \mathit{BuiltInGroups}$, the algorithm simply returns $\mathtt{R}(B, \mathit{Elts}(g))$, which we have according to our assumptions. Finally, if P is any other group name g (so, a group name for which no definition or implementation is available), the algorithm returns $\emptyset$ (for $\mathtt{R}_\Downarrow$) or $\mathtt{S}(B)$ (for $\mathtt{R}_\Uparrow$).

The computation of $\mathtt{R}_X(B, P)$ basically amounts to parsing B, top-down, as an element of the formal language associated with P. It is common for top-down parsing not to work, or not to work well, when any grammar productions are left-recursive (of the form $g \to g \ldots$ where g is a non-terminal). Here, left-recursion could cause the algorithm to fall into an infinite loop. In theory, left-recursive productions can always be avoided (in particular, by using Greibach normal form). In our setting, however, we do not wish to restrict or to rewrite group definitions in order to prevent left-recursion.

Therefore, we prefer weakenings of the definition of $\mathtt{R}_{\Downarrow}$ and $\mathtt{R}_{\Uparrow}$ that work without the assumption. For a conservative implementation, we require only:

$$\mathtt{R}_{\Downarrow}(B, P) \subseteq \mathtt{R}(B, \mathtt{Meaning}_{\rho_{\Downarrow}}(P))$$
$$\mathtt{R}_{\Uparrow}(B, P) \supseteq \mathtt{R}(B, \mathtt{Meaning}_{\rho_{\Uparrow}}(P))$$

Fortunately, it is not hard to adapt our algorithm to achieve these properties while improving its efficiency and guaranteeing its termination. In particular, we can allow calculations to terminate—with a conservative decision—whenever a given computational budget has been exhausted. As a special case, we can allow queries on servers to time out. Furthermore, by passing an additional argument to $\mathtt{R}_{\Downarrow}$ and $\mathtt{R}_{\Uparrow}$, we can keep track of the set of groups that we have examined, and terminate—again, with a conservative decision—when we detect a loop. We have studied variants that detect all loops or only those loops that arise as a result of left-recursion. Only the latter loops cause divergence, but the former variant is a little simpler and, we expect, adequate for our purposes. (We omit lengthy details on this point.)

Writing $\mathtt{R}_{\Downarrow}(B, P)$ and $\mathtt{R}_{\Uparrow}(B, P)$, respectively, for these approximations of $\mathtt{R}(B, \mathtt{Meaning}_{\rho_{\Downarrow}}(P))$ and $\mathtt{R}(B, \mathtt{Meaning}_{\rho_{\Uparrow}}(P))$, we obtain a conservative implementation of `IsAuthorized`:

$$\mathtt{IsAuthorized}^{\mathrm{imp}}(M, A) = \exists B \in M.\mathtt{IsAuthorized}_1^{\mathrm{imp}}(B, A)$$

$$\begin{array}{l}\mathtt{IsAuthorized}_1^{\mathrm{imp}}(B, A) = \\ \quad \mathtt{case}\ A\ \mathtt{of} \\ \quad\ \mathtt{empty} : \mathtt{false} \\ \quad \mid\ A',\ \mathtt{Allow}\ P : \mathtt{R}_{\Downarrow}(B, P) \neq \emptyset \vee \mathtt{IsAuthorized}_1^{\mathrm{imp}}(B, A') \\ \quad \mid\ A',\ \mathtt{Deny}\ P : \mathtt{R}_{\Uparrow}(B, P) = \emptyset \wedge \mathtt{IsAuthorized}_1^{\mathrm{imp}}(B, A')\end{array}$$

Thus, we replace occurrences of `R` with $\mathtt{R}_{\Downarrow}$ for `Allow` checks and with $\mathtt{R}_{\Uparrow}$ for `Deny` checks.

5.4 Distribution

When ACLs and groups are defined in terms of other groups, it remains to spell out how the corresponding servers contribute to an ACL check. This process may be orchestrated by the client that requests access or by the entity that holds the ACL. For example, if the ACL refers to a group g_1 which is itself defined in terms of a group g_2, the client may obtain and present evidence about g_1 and g_2, or the

entity that holds the ACL may do the lookups for both groups. Alternatively, this entity may contact a server responsible for g_1, which in turn may contact a server responsible for g_2.

It is this alternative scheme that we adopt as our primary one:

- the evaluation of $\texttt{IsAuthorized}^{\text{imp}}(M, A)$ happens locally at the entity that holds A, with calls to others for evaluating $\texttt{R}_X$;
- the evaluation of $\texttt{R}_X(B, P)$ uses local recursive calls in all cases indicated by the definition of $\texttt{R}_X$, except in the case of $\texttt{R}_X(B, g)$, for which a server responsible for g should be consulted (unless, as indicated above, this would cause looping).

In practice, this scheme can be subject to many optimizations, such as caching, batching of queries, and "pushing" of credentials by clients (e.g., [5]).

With this scheme, ACLs and the group memberships are partly revealed only in response to queries ($\texttt{IsAuthorized}^{\text{imp}}$ queries for the ACLs, $\texttt{R}_X$ queries for the groups). An observer who can see enough message flows may also infer dependencies, namely that particular ACLs or groups depend on certain other groups. However, the full contents of ACLs and groups are not disclosed wholesale.

Without atomicity assumptions, it is possible that group definitions are changing during the evaluation of $\texttt{IsAuthorized}^{\text{imp}}(M, A)$. For example, let A be the ACL `Allow Friends, Deny Friends`, and suppose that a member `Alice` is being added to the group `Friends`. If the addition to the group happens between the processing of the two clauses of the ACL, $\texttt{IsAuthorized}^{\text{imp}}(\texttt{Alice}, A)$ will return `true`, a behavior that could happen neither before nor after the addition. We have considered techniques that prevent this behavior. One of them consists in asking the servers responsible for the relevant groups to provide information current as of the time of the ACL check of interest, via an extra "time" parameter for $\texttt{R}_X$. Assuming that the servers keep a log of recent group changes, this technique would help for ACL checks that complete reasonably fast, subject to the limitations of clock synchronization. Whether such techniques are in fact necessary remains open to debate.

6 On Prefix Matching

In this section we elaborate on prefix matching, described in Section 4, and explain why we do not adopt it. Our reasons have to do with `Deny` clauses and groups; they are weaker if either of those features is absent. Since we believe that prefix matching is not essential for expressiveness or usability, we opted to omit it in order to give a better treatment of those features.

The rationale for prefix matching is as follows. Suppose that a blessing B' matches an ACL and that $B \preceq B'$. Whoever holds B can extend it to B', thus passing the ACL check; therefore, not letting B match the ACL may cause inconvenience and has no immediate benefit if B behaves maliciously. (It may however protect against accidental misbehavior.)

Adopting prefix matching would mean, for example, that the ACL

$$\texttt{Allow}\ n_1\ \texttt{/}\ n_2$$

grants access when the blessing n_1 is presented. Beyond this trivial example, it is less clear what to do in other situations.

Let us consider the ACL

$$\texttt{Allow}\ n_1\ \texttt{/}\ g$$

and imagine that g is defined to be empty. Should access be granted when the blessing n_1 is presented? A positive answer would seem rather surprising, and is not justified by the proposed rationale for prefix matching: there is no way to extend n_1 so that it matches n_1 / g exactly. Prefix matching for a blessing pattern P (n_1 / g in this example) is about the prefixes of the blessings that match P, not the blessings that match the prefixes of P. In other words, it is about the prefixes of the meaning of P, not the meaning of the prefixes of P.

Next let us consider the ACL

$$\texttt{Allow}\ n_1\ \texttt{/}\ n_2,\ \texttt{Deny}\ n_1\ \texttt{/}\ n_2$$

Should access be granted when the blessing n_1 is presented?

- We could answer this question positively by computing the meaning of the `Allow` clause (which, with prefix matching, implies authorizing n_1), the meaning of the `Deny` clause (which does not imply rejecting n_1), and then taking the difference. This behavior seems odd, and is not justified by the proposed rationale for prefix matching: there is no way to extend n_1 so that it matches n_1 / n_2 but does not match n_1 / n_2.
- An alternative approach consists in computing all the blessings allowed by the entire ACL (subtracting for `Deny` clauses, but without prefixing for `Allow` clauses), and then adding all their prefixes. As the example illustrates, subtracting for `Deny` clauses does not commute with adding prefixes. This alternative approach does conform to the rationale for prefix matching.

Unfortunately, the alternative approach appears difficult at best. Let us consider the ACL

$$\texttt{Allow}\ n_1\ \texttt{/}\ g_1\ \texttt{/}\ \texttt{eob},\ \texttt{Deny}\ n_1\ \texttt{/}\ g_2\ \texttt{/}\ \texttt{eob}$$

where g_1 and g_2 are group names and `eob` is our special terminator name. According to the alternative approach, access should be granted when the blessing n_1 is presented if and only if there is some element of g_1 that is not in g_2. In the general case where group definitions may contain cycles, we face the inclusion problem for context-free languages, which is undecidable! Even without cycles, we do not have a satisfactory solution. Straightforward algorithms that require enumerating the members of g_1 or the non-members of g_2 seem unattractive from efficiency and privacy perspectives.

7 Conclusion

As noted in the Introduction, many systems support distributed authorization. Generally, their features include groups; sometimes, they also include forms of negation, and more rarely compound names and local names. There is no canonical solution to problems such as missing, unavailable, and circular group definitions, which are made more delicate by negation and compound names. The pioneering article on Digital's DSSA noted that "it is impractical, in a distributed environment where group nonmembership cannot be certified, to implement denial to arbitrary groups" [4]. Years later, SDSI allowed an operator `NOT` on groups, requiring certificates of non-membership. In SDSI, the fundamental algorithm for checking group membership worked entirely locally, by computing on credentials; in contrast, we describe a distributed algorithm.

A salient aspect of our design, which mitigates those difficulties, is that ACLs contain negative clauses but groups do not, and that ACLs cannot be reused for defining groups or other ACLs. This choice also enables us to provide a liberal semantics for ACLs (in which, for example, the ACL `Allow Alice` permits access with `Alice / Phone`) distinct from that of groups (according to which a group that contains `Alice` need not contain `Alice / Phone`). The semantics of ACLs contrasts with the treatment of compound principals in previous work (e.g., [5, 7, 8]). There, an ACL that would grant access to `Alice` would generally not automatically grant access to a compound principal of the form `Alice op Phone`, where `op` is a binary operator, unless this operator happens to be conjunction ($\wedge$). Conjunction hardly resembles `/`, for example because it is commutative; other operators previously considered seem closer to `/`. Beyond these differences, the fact that we have only one operator (`/`) and that it is associative allows us to sharpen the helpful connection with formal languages.

The realization of our design is under way. While the design addresses expressiveness and semantic questions with some consideration for implementation strategies, its realization may rely on a number of optimizations, such as caching. It may also lead to the development of auxiliary tools and idioms; in particular, further work on conventions and on grouping objects could be helpful in writing and managing policies.

Acknowledgments We are grateful to Cosmos Nicolaou and to Jiří Šimša for helpful comments on drafts of this paper.

References

1. Birgisson, A., Politz, J.G., Erlingsson, Ú., Taly, A., Vrable, M., Lentczner, M.: Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In: 21st Annual Network and Distributed System Security Symposium (2014)
2. Bodei, C., Degano, P., Focardi, R., Priami, C.: Authentication via localized names. In: Proceedings of the 12th IEEE Computer Security Foundations Workshop, CSFW. pp. 98–110 (1999)

3. Cramer, R., Damgård, I.: Multiparty computation, an introduction. In: Contemporary Cryptology, pp. 41–87. Advanced Courses in Mathematics – CRM Barcelona, Birkhäuser, Basel (2005)
4. Gasser, M., Goldstein, A., Kaufman, C., Lampson, B.: The Digital Distributed System Security Architecture. In: Proceedings of the 1989 National Computer Security Conference. pp. 305–319 (1989)
5. Lampson, B., Abadi, M., Burrows, M., Wobber, E.: Authentication in distributed systems: Theory and practice. ACM Transactions on Computer Systems 10(4), 265–310 (1992)
6. Lampson, B.W.: Computer security in the real world. IEEE Computer 37(6), 37–46 (Jun 2004)
7. Rivest, R.L., Lampson, B.: SDSI — A Simple Distributed Security Infrastructure (October 2, 1996), version 1.1, at `http://theory.lcs.mit.edu/~rivest/sdsi11.html`
8. Wobber, T., Yumerefendi, A., Abadi, M., Birrell, A., Simon, D.R.: Authorizing applications in singularity. In: EuroSys '07: Proceedings of the 2007 conference on EuroSys. pp. 355–368. ACM Press, New York, NY, USA (2007)