DRACL (Decentralized Resource Access Control List)

by Steven D. Allen

S.B, C.S. M.I.T. 2014

Submitted to the

Department of Electrical Engineering and Computer Science in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering in Electrical and Computer Science

at the

Massachusetts Institute of Technology

August 2016

Author: ____

Department of Electrical Engineering and Computer Science
August 31, 2016

Certified by: ____

David Karger, Professor, Thesis Supervisor

Certified by: ____

Vinod Vaikuntanathan, Professor, Thesis Supervisor

Accepted By: ____

Dennis M. Freeman, Chairman, Masters of Engineering Thesis Committee

DRACL (Decentralized Resource Access Control List)

Author: Steven Allen *Advisor:* David Karger

August 31, 2016

In Partial Fulfillment of the Requirements for the Degree of Master of Engineering in Electrical Engineering and Computer Science.

Abstract

DRACL is a privacy-preserving, scalable, secure, and developer and user friendly federated access control system. It allows producers to manage, through a single authentication provider, which consumers can access what content across all content hosts that support the DRACL protocol. It preserves user privacy by not revealing the producers' social networks to content hosts and consumers and allowing content consumers to access content anonymously. Unlike existing solutions, DRACL is federated (cf. Facebook Connect, Google Sign-In), does not have a single point of failure (cf. Mozilla Persona, OpenID), and does not reveal its producers' social networks to content hosts (cf. Facebook Connect's `user_friends` permission).

Contents

Chapter 1

Introduction

Today, the web is basically a big social network. Producers — users who post content — post content on various content hosts (e.g., Facebook) while consumers — users who access content — interact with this content on these various content hosts. This really is an amazing system that allows people to interact socially across barriers that were previously insurmountable (e.g. distance, culture, and social status).

Unfortunately, these content hosts don't interoperate well and this lack of interoperability introduces barriers to social interaction through both poor usability and privacy issues. Poor usability directly bars communication by making it harder. Privacy issues bar communication by discouraging users who value their privacy from socializing on the web and encouraging those who do socialize to self-censor.

While all users could simply sign up for the same monolithic service, this is inherently bad for privacy. The monolithic service can do pretty much whatever it wants and get away with it because there is no competition. Therefore, users will either self-censor or choose to simply not socialize online. Unfortunately, this is what we have today with Facebook.

On the other hand, if users choose to use diverse content hosts, the lack of interoperability becomes a social barrier. For one, consumers need to have accounts on every content host used by their friends. This is a usability problem because it forces users to manage multiple accounts and a privacy problem because content hosts often require that account holders sign up with their "real" names. Additionally, producers need to recreate their social networks on every content host. This is also a usability problem because it forces producers to manage their "friends list" in multiple locations and is also a privacy problem because it forces producers to expose their "friends list" to multiple services.

In this thesis, we propose a privacy-preserving, scalable, secure, user and developer friendly web-wide federated access control scheme that solves a significant portion of this problem. Our system allows producers to share content with consumers without having to manually recreate their “friends list” on every service they use regardless of what content hosts their friends use. Furthermore, it allows consumers to anonymously access any content shared with them on any content host.

1.1 The Big Picture

In this section, we look at the bigger picture to understand where our access control system fits into our dream of how the web should look.

1.1.1 The Dream

The web should look like one big interconnected social network. People should be able to share with anyone on any host regardless of whether or not said person has an account with any specific host. Additionally, people should be able to access material shared with them without having to jump through hoops, sign up, or hand out personal information to the website hosting the content. It should be a web without walls.

It should be user friendly. People shouldn't have to manage multiple accounts or manually re-create their social network on every service they use.

It should be free and fluid. People should be able to choose what services they use, where they host content, and how they communicate. They should be able to move from service to service and even communicate across multiple services without fragmenting their social networks.

It shouldn't require users to sacrifice their privacy in order to participate. People should be able to access content shared with them without being forced to give up personal information. People should be able to share content without revealing their social network to the party that happens to host the content.

It should be secure and robust. A single service going down shouldn't shutdown social communication online. A compromise of a single service shouldn't lead to a cascading compromises of other services.

1.1.2 Reality

The web does not look like one big interconnected social network. Instead, it's filled with walled off "social networks" that don't interoperate by design.

It's not user friendly. Users must manage multiple accounts. Some smaller services give users the option to authenticate with one of a few "blessed" third-party accounts but these aren't standardized and don't give the user the freedom to choose who they trust (they have to choose from a small set).

It's not free and fluid. Someone on Google+ can't share content with someone on Facebook. This fragments social networks and forces people to stick with services they may not like simply because that's where their friends are.

It impossible to socialize with friends and family online without sacrificing privacy. For example, Facebook requires anyone wanting to simply access content to have an account and requires that all account holders sign up with their "real name". On a web without walls, people should be able to access content shared with them on, e.g., Facebook without ever signing up or telling Facebook anything about themselves.

It's not secure and robust. Shutting down Twitter and Facebook effectively shuts off most social interaction on the web. A compromise of Facebook would compromise most semi-private social interactions on the web and would compromise all websites that use Facebook's authentication service, Facebook Connect.

1.1.3 Dissecting The Problem

We've broken this dream into four building blocks: content hosting, identity management, access control, and content notification. Here, we only tackle access control but it helps to understand the bigger picture.

1.1.3.1 Content Hosting

Content hosting addresses the problem of storing content and social media interactions. To interact online, people need a place to store their interactions. That's the content host's job.

The problem of hosting content has been solved; this is the web as-is. People can already choose from a plethora of content hosts such as Facebook, Flickr, Google+, Twitter, Google Docs, etc. The existing solutions aren't always perfect: it's still very hard for people to move their content from one content host to another. However, this is slowly being fixed by data export tools (e.g., Facebook's "Download Your Information" service).

1.1.3.2 Identity Management

Identity management addresses the problem of naming someone on the internet. Identity management is a prerequisite for any social network that isn't entirely public (e.g. 4chan) because a producer needs to be able to name, or describe, a consumer to state whether or not that consumer should be able to access something. In our case, we need an identity management solution that allows naming people across services.

Many systems attempt to address the identity management problem. These include Facebook Connect, Keybase, NameCoin, and GnuPG (OpenPGP). Unfortunately, none of these are perfect: Facebook is completely centralized, Keybase is somewhat centralized, NameCoin is untested and complicated, GnuPG's user experience is needlessly complicated (it usually requires manually comparing long key fingerprints and mucking around with arcane command invocations), and none of the decentralized systems have wide adoption. However, this *is* a well explored problem lacking only a simple, widely adopted decentralized solution; probably because there isn't much use for one at the moment outside of sending encrypted email.

1.1.3.3 Access Control

Access control addresses the problem of limiting who — identified by the identity system — can access what. To make the web look like one big social network, we need an access control system that can limit access to content independent of where it is stored.

There really aren't any systems that generically address the problem of access control problem. For example, there's no way to tell Flicker "allow my classmates to access this photo album" unless Flicker has an explicit list of your classmates and they all have accounts on Flicker. There are federated social media platforms like Diaspora and that allow this kind of cross-platform interoperability; however, these kinds of systems usually attempt to solve all four problems at once. This means they *can't* be used with any of the existing content hosts without requiring extensive modification to these content hosts. This would require re-building the web from scratch and throwing away what we already have.

1.1.3.4 Content Notification

Content notification (or publish-subscribe) addresses the problem of consumers learning about content they can access. Conversely, it allows publishers to notify consumers about content. We need this because, while an access control system can permit or deny a consumer's access to content, it doesn't actually tell the consumer where to find content they can access. In Facebook terms, this is the *status/feed*. To achieve our goal, we need a content pub-sub system that allows producers to advertise content to consumer independent of where it is stored or where those consumers have accounts.

There are some pretty decent decentralized web-wide pub-sub systems although many of these systems aren't widely adopted. The current most widely adopted decentralized solution is email — although email the "sub" part of pub-sub doesn't really apply to anything but mailing lists. However, people tend to get more email than they can handle anyways. Better solutions are systems like RSS or even distributed Twitter-like systems such as pump.io[6].

1.1.4 Our Contribution

In this thesis, we address the access control problem. We design a privacy-preserving, scalable, secure, user and developer friendly access control system for the web that allows producers to manage groups of consumers and declare which groups can access what content what content across all content hosts that use DRACL through a single authentication provider chosen by the producer. Additionally, our system preserves privacy by hiding the producer's social network from both content hosts and consumers and it by allowing consumers to access content anonymously.

We build our access control system on top of the web as it is. Therefore, we can use the existing rich ecosystem of content hosts without having to re-build the web from scratch.

We assume that the web will eventually standardize on a single identity management system. We've designed our access control system to work on top of any identity system as long as this identity system is able to map identities to asymmetric cryptographic keys. Keybase has demonstrated that pretty much all identity systems can do this in some form or another through the use of special cryptographic proofs.

We also assume that the web will eventually standardize around some reasonably small set of content notification protocols. There are plenty to pick from, it's just a matter of adoption.

While we're still a ways off from a web without walls, we provide a large missing component and were able to do so without redesigning the web from scratch.

1.2 Terminology

Before actually diving into the problem of access control and our solution, we need a consistent, well defined language for discussing it. Below, we define some critical terminology used in this document.

Identity An assumed identity. That is, identities may not correspond one-to-one to real people. They may be pseudonymous, or may correspond to groups of people.

Producer A user that publishes content and wishes to control access to said content.

Consumer A user that accesses content.

Authorized Consumer A consumer that is authorized to access a particular resource.

Group A group of users as defined by a producer. In DRACL, groups are the unit of access control (i.e., producers grant access to groups, not directly to individual consumers).

Authentication Provider (AP) A helper service for facilitating access control. Every DRACL producer will have an AP (just like every email user has an email provider). Basically, the AP can perform limited actions on behalf of producers while they're offline and can help ensure that a compromise of a producer's account is recoverable. We assume that the AP isn't actively malicious but can be compromised and may be nosy.

Content Host A website/service using DRACL to authenticate content it hosts. For example, Flicker, Facebook, Imgur, etc.

Friend Same as a Facebook friend. That is, the relationship may or may not be friendly.

Honest But Curious Party An honest but curious party follows protocols as specified but attempts to learn information they shouldn't know by looking at the protocol's trace. That is, they aren't actively malicious, just nosy. This is basically the standard cryptographic definition of honest but curious.

Malicious Party A malicious party, on the other hand, will deviate from the protocol when convenient.

1.3 Requirements

We would like a privacy-preserving, user and developer friendly, secure, and scalable access control system for the web that promotes user freedom. In this section, we discuss what these requirements mean in the context of access control systems and why we care about them. We make no assumptions about whether or not such a system already exists; we simply state what we want out of such a system.

Two explicit non-requirements are hiding the content from content hosts and hiding metadata — who's communicating with whom — from global adversaries such as governments. These goals are simply out of scope of an access control system. The first can be provided by simply encrypting the content and the second and can be provided by a dedicated metadata-hiding anonymous communication system like Vuvuzela [30]. If you're looking for a system that provides this functionality you're lost, look elsewhere.

We also expect that (authorized) consumers and producers will be honest but curious, not actively malicious. At some level, discouraging "bad" behavior becomes a social problem, not a technical problem.

1.3.1 Privacy Preserving

By privacy preserving, we mean that a good access control system should avoid revealing metadata. That is, it should reveal neither the identities or actions of consumers nor the composition of its producers' social networks to any party when possible.

First, such a system should not reveal the identities of consumers to content hosts because people shouldn't be excluded from social interactions for valuing their privacy. In the web as it exists today, it's impossible to participate in social networks without handing out personal information to third parties. This forces people to either hand out this information or isolate themselves completely. While, to prevent abuse, social networks often need producers/producers to identify themselves, we believe that consumers should be able to access content without identifying themselves. This would allow people to anonymously access content on any content host while only publishing content on content hosts they trust. While many consumers may choose to identify themselves (e.g. by logging in), this should be by choice.

Additionally, such a system should also not reveal the identities of consumers to content hosts because it would discourage the sharing and consuming of content due to fear of judgment by association. Producers may choose to not share a piece of content with a consumer because they don't want to be associated with the consumer. Consumers may choose to not access a piece of content because they don't wish to be associated with the producer. In general, a good access control system should not influence what people share.

Second, such a system should not reveal consumer activity to anyone, especially producers. If a consumer views a piece of content, the content's producer should not be notified in any way without the consumer's explicit consent. Otherwise, the consumer loses the freedom to choose to either defer or avoid participation in a social interaction. For example, if the fact that a consumer has read an invitation were revealed to a producer, the consumer would be pressured to out-right reject or accept the invitation instead of simply pretending not to have seen it. This would force the consumer to either lie ("I have plans"), do something socially unacceptable (reject it), or attend some event they don't want to attend (potentially dangerous). Basically, how to respond to a piece of content should be up to the consumer. However, we don't believe that it's the access control system's job to prevent actively malicious producers from subverting the protocol to learn this information. Such behavior should be discouraged through social means and such producers are likely to have a hard time making friends.

Finally, such a system should not reveal the structure of a producer's social network, *especially* to members of their social network, because would again influence if and to whom a producer chooses to share content. People should feel free to exclude others from their social network; the ability to communicate privately (exclude unwanted parties) is necessary to prevent self-censorship. For example, if an access control system revealed that two consumers are members of some shared group while a third consumer is not, a producer might feel pressured include all three in the same group to avoid offending the third. This would, in turn, cause the producer to either share content more widely than they might otherwise choose or refrain from sharing at all (a chilling effect).

Privacy is an essential freedom for enabling the free expression of social behaviors. Any information disclosure in a system limits the usefulness of that system to social behaviors that can tolerate the disclosure of that information. A web-wide access control system that doesn't protect user privacy would therefore inhibit the types of social behavior applicable to the web. This is not to say that all social behavior should be tolerated. However, it's not the job of an access control system to limit social behavior.

1.3.2 User Friendly

By user friendly, we mean that sharing and accessing content across content hosts should be as easy as sharing content on a centralized social network like Facebook. Users shouldn't have to manage multiple sets of credentials or manually recreate their social network on every content host they use.

Currently, content hosts (usually) force their users to manage one set of credentials per service. In addition to being beyond annoying, this practice encourages users to choose simple passwords or reuse them. A good global access control system can eliminate this problem by allowing users to manage a single access control account and re-use the same account from service to service.

Additionally, an access control system shouldn't assume that producers will run their own servers or pay for anything. The amount of invasive advertising, poor software, and privacy violations people tend to put up with on the internet for "free" services is a testament to how far people will go to avoid paying.

Finally, a user friendly access control system needs to support groups. Groups allow producers to manage access to content at a level more coarse grained than individual consumers but finer grained than "all friends". Without support for groups, producers wishing to share with more than a few consumers are likely to just share the content with all of their friends because manually selecting more than a few users is extremely tedious.

Group support goes deeper than the UI. Adding/removing a consumer to/from a group should retroactively grant/revoke that consumer's access to the group's content. Otherwise, producers would have to manually grant/revoke access on a content-by-content basis which would be extremely tedious. Worse, it's error prone so a producer might forget to revoke a malicious consumer's access to some piece of content.

Again, this comes back to not limiting social interaction on the web. Usability problems in an access control system turn into communication barriers because users will choose not socialize in some way because it's hard. Therefore, a good access control system should try to stay out of the user's way as much as possible and make socializing online as easy as possible.

1.3.3 User Freedom

By user freedom, we mean that a good access control system should enable users to choose which online services they use and should allow them to freely to move between them. This is closely related to both privacy and user friendliness as it's a prerequisite for both but it bears discussing as a stand-alone category.

Currently, it's prohibitively difficult to move between content hosts because because social networks are usually tied to a single content host. A global access control system can alleviate this problem by allowing producers to grant access to their friends on any content host regardless of what content hosts their friends use.

A consequence of making it easier to move between content hosts is more practical competition as it lowers the barrier for bootstrapping new services. This is a direct consequence of allowing producers to share content with consumers without forcing these consumers to create accounts. Competition is not only good for users, it's absolutely necessary to keep companies from taking advantage of users in pursuit of higher profit margins. See AT&T's [17] policy and Comcast's [18] recent attempt to charge an additional fee for not snooping on their customers browser history.

Finally, a good access control scheme must be decentralized. We have an entire section on this later (see § 1.4.1) but the short version is that decentralized systems give users the ability to choose who they trust. Forcing users to use the same authentication system would defeat the purpose of allowing users to choose and move between content hosts. The centralized authentication system would become the new Facebook.

1.3.4 Secure

By secure, we mean that only authorized consumers should be able to access content, producers should be able to efficiently and reliably revoke access from consumers, and that compromises should be recoverable.

First, only authorized consumers, the content's producer, and the content host should be able to access content — no "trusted" third parties. Currently, if someone were to hack Gmail, they'd be able to access pretty much everything on the public internet (through, e.g., password reset emails). This was a mistake. Therefore, a good access control system should avoid this problem by design.

Second, for efficient and reliable revocation, a producer should not have to contact every content host in order to remove a consumer from a group. First, this would trigger a lot of up-front network traffic, even for old content that may never be accessed again. Second, it would violate the fail-safe principle of systems design: if a producer were unable to contact some of their content hosts for some reason, there would be no way to revoke access (assuming that the content host comes back online eventually).

Finally, no compromise of any single party should be unrecoverable and no compromise of any single party should bring down the entire system. Websites are hacked all the time and user account compromise is common so any system that can't recover from compromise is dead in the water.

Users must feel safe socializing online. They must be able to rely on the fact that even if something goes slightly wrong — which it will — their online interactions will still, for the most part, be private and secure.

1.3.5 Developer Friendly

By developer friendly, we mean that a good universal access control system should be at least as easy to deploy as a custom password-based identity scheme — the current de facto standard — and it shouldn't require significant infrastructure changes.

We believe that a lack of developer friendliness contributed significantly to the failure of systems like OpenID[4]. More concretely, we know that the difficulty to deploy and maintain SSL certificates has been a significant barrier to adoption of SSL as demonstrated by the rapid rise — 5% in 6 months — of SSL deployment after the launch of Let's Encrypt [13].

One of the key benefits of abstractions like access control systems is that they tend to make developer's lives easier. Instead of having to reinvent the wheel at every turn, a developer can just delegate the problem to a ready-made solution. A good access control system should take advantage of this to fuel its adoption.

1.3.6 Scalable

By scalable, we mean a good web-wide access control system should scale to work for the entire web without imposing unreasonable uptime, bandwidth, security, or computation requirements on any party.

It can't assume that anyone can build a perfect implementation. No complex system can hope to achieve perfect security and perfect uptime and errors become more probable at scale (a fact of probability). At the end of the day, this system will be built and maintained by humans so we must design it with that in mind.

Such a system should keep any work done by any third parties to a minimum. This is a direct result of users not being willing to pay for anything: nobody can't afford to run overly expensive computations on behalf of users for free or provide infinite storage or bandwidth. Additionally, the better an access control system scales, the less money it takes to run it and the more likely it is for a benevolent third party to be able to do so free of charge.

1.3.7 Summary

We've covered a lot in this section however, all this discussion really boils down to the following requirements:

1. The producer's social network is private, even from friends.
2. With whom a producer shares a piece of content is private, even from friends.
3. The consumer should be able to access content anonymously.
4. Supports usable group management.
5. Requires consumers to manage at most one set of credentials.
6. Enables user-choice.
7. Is easy to deploy on content hosts.

8. No single point of failure or bottleneck.
9. All compromises are recoverable.
10. Only permits explicitly authorized access.

1.4 Existing Systems

Given the requirements stated in the previous section, this section explores existing systems and motivates the need for a new one.

1.4.1 Identity-Based Authentication Systems

The most common access control scheme is to assign one or more identities to all users, give them a way to authenticate against these identities (prove they are someone), and then have individual content hosts implement custom access control systems on top of the identity-based authentication system. In other words, there *is* no global access control system.

This type of system is inherently bad for both usability and privacy. They're bad for usability because they provide no way to centrally manage groups of users. They're bad for privacy because content hosts learn the identities of who has access to what and, by extension, who knows who. While these issues sufficiently motivate an alternative design, we have included this section to give an overview of some common existing systems and to learn from their shortcomings.

1.4.1.1 Site Specific Authentication

The vast majority of content hosts today require users to create site-specific accounts. This is a poor solution to the access control problem for both users and content hosts because it introduces security hazards and has poor developer and user usability. The only goal this system meets is that no unauthorized third party can access a producer's content. It's also arguable that it meets the goal of not revealing the producers social network to consumers but this isn't guaranteed and content hosts don't always get this right. For example, in 2010, Google launched a social network called Google Buzz and automatically created accounts for its current Gmail users. Unfortunately, they decided to list every user's most frequent contacts on their *public* profile page [11].

From a user usability standpoint, site-specific accounts force users to create new accounts and replicate their social networks on every content host they use. As discussed in the requirements section on user friendliness (§ 1.3.2), this is bad for usability.

From a user-freedom standpoint, site-specific accounts make it hard to switch between services because these service simply don't interoperate. Users can't just switch to a new content host because they have to convince their friends to create accounts on the new content host before they can make use of it.

From a developer usability standpoint, site-specific accounts force content hosts to implement custom account/access control systems. Again, refer to the requirements (§ 1.3.5) for why this is a problem.

Site specific accounts are a security hazard because content hosts are notoriously bad at safely storing credentials and users are notoriously bad at choosing/remembering safe passwords. For example, content hosts often store user credentials in the clear [5] and users often reuse passwords and/or use weak passwords [21].

To be fair, site-specific accounts have served the web well for a very long time. This is primarily because they have no external dependencies, are *really* easy to implement in modern web frameworks, and will continue to work years down the road no matter how much the web changes. However, we still believe that their many problems warrant a better solution.

1.4.1.2 Centralized Identity-Based Authentication

Centralized identity-based authentication systems, such as those provided by Google and Facebook, allow users to authenticate as an identity to multiple sites using a single set of credentials. These systems therefore solve all of the security problems we mentioned in the site-specific authentication section as users only have one set of credentials managed by a single, hopefully competent, entity. They also solve the associated usability problem of users having to remember multiple passwords.

Some centralized identity-based authentication systems improve usability by allowing users to carry their social networks with them to content hosts. For example, Facebook Connect allows content hosts to access to users' friends lists by requesting the `user_friends` permission[1].

Unfortunately, centralized identity-based authentication systems can't provide a way to do so while hiding the producer's social network from content hosts by definition. That is, for one user to allow another user to access a resource on a content host, the first user must identify the second user to the content host. This is a fundamental problem with identity systems because they operate on the level of identity and don't provide an access control system.

Again, because these systems operate on the level of identity, they force content hosts to implement their own access control systems (and the associated data models, user interfaces for managing access, etc.) which, again, violates our stated requirements (§ 1.3.5). Content hosts shouldn't have to reinvent the wheel.

Additionally, because these systems are centralized, users are forced to choose between a few "accepted" providers and can't run their their own. This is a problem for user freedom because it doesn't allow free competition between authentication providers.

Finally, content hosts are forced to support whatever authentication systems happen to be in vogue at the time. Even though many centralized identity systems use standards like OAuth 1.0 [23] to make integration with content hosts easier, they must still be blessed on a one-by-one basis.

1.4.1.3 Decentralized Identity-Based Authentication

Decentralized identity-based authentication protocols such as OpenID [4], Persona [2], and WebID [10] allow users to identify to multiple services using the same credentials but, unlike centralized identity-based authentication systems, decentralized identity-based authentication protocols provide better user freedom by not forcing users to choose between a few "accepted" providers. Additionally, in theory, content hosts should be able to support exactly one decentralized identity-based authentication protocol — assuming they eventually converge.

While decentralized identity-based authentication protocols address the user choice concern, they still don't address our privacy concerns because they still operate on the level of identity.

1.4.2 Access Control Systems

Unlike identity-based authentication systems, access control systems directly dictate what operations a system should and should not permit. Where an identity-based authentication system answers the question "Does PROOF imply that CLIENT is IDENTITY?", access control systems answer the question "Does PROOF imply that

CLIENT has PERMISSION?”. In our case, “PERMISSION” is usually “can access CONTENT”. This is actually just a generalization of identity-based authentication systems; in a permission system, “PERMISSION” is simply “is IDENTITY”.

This categorically addresses our usability concerns with identity systems because the entire social network is now defined in a single system (the access control system) instead of spread out across various content hosts. Unfortunately, there aren’t any existing access control systems that meet our privacy and security requirements.

1.4.2.1 Centralized Access Control

Centralized access control systems such as Kerberos [26] and LDAP [24] allow services to offload user/group management to third parties (the authentication provider). While, as noted above, this solves the usability problems in centralized authentication systems, it makes the user-freedom problem much worse. While content hosts can allow different users to authenticate with different centralized authentication services, they cannot allow users to choose their own (fully) centralized access control service without partitioning the social network because, by definition, centralized access control services don’t interoperate. Note: If a set of semi-centralized access control systems were built on top of some decentralized authentication system, they could interoperate to an extent but (1) no such system exists and (2) this system would still be limited to blessed parties.

1.4.2.2 Decentralized Access Control

Decentralized (federated, really) access control systems offer the same benefits as centralized access control systems but without the drawbacks of being a centralized system. That is, users can freely choose between providers or run their own. Thus, decentralized access control systems categorically meet our user friendliness requirements and help us meet our user freedom requirements.

There are many existing cryptographic protocols for decentralized access control such as Decentralized Access Control with Anonymous Authentication of Data Stored in Clouds [28]. Unfortunately, these are simply cryptographic protocols and don't attempt to provide a full system specifications. Furthermore, we found it difficult to build an actual system around these protocols because they were designed in a vacuum without any thought to real-world applicability (usability, fallibility, and efficiency). For example, revocation in [28] requires updating every access control list that mentions the revoked, lets users learn their "attributes" (the groups they're in), and provides no mechanism for recovering from compromise.

On the other hand, there are real decentralized access control system design such as DRBAC [22], Rule-Based Access Control for Social Networks (RBACSN) [19], and the Kerberos Consortium's User Managed Access (UMA) [9]. Unfortunately, none of these protocols even consider privacy; it simply doesn't factor into their designs. They never try to ensure consumer anonymity nor do they try to fully hide the producer's social network from consumers. Furthermore, UMA and DRBAC both require their users to completely trust some AP with access to all their content; if the AP is snoopy, it could access all it's user's content. Worse, compromising an insecure AP would compromise the entire system. On the other hand, RBACSN doesn't allow restricting access to groups or individual consumers; that is, it allows producers to share content based on their relationship with consumers (direct friend, friend-of-a-friend) etc. but doesn't support simply creating groups of friends and sharing content that group of friends.

Now we're in uncharted territory. So far, we've decided that we need an access control system, not just an authentication system, and would like it to be decentralized. Next, we'll delve into the decentralized access control design space and work our way towards a design that satisfies our requirements.

1.5 Design

Now that we've determined that we need a decentralized access control protocol, we explore the design space and work towards the final design of DRACL. If you just want to know how DRACL works but not why or how we got there, proceed to the System Overview (chapter 2) for a top-down view of the DRACL protocol.

Note: *The straw men presented here are not complete solutions. We highlight the problems we feel are important enough to motivate moving to a new solution but gloss over many* ***dangerous*** *security flaws. Please* ***do not*** *attempt to implement any of the straw men presented in this section without serious analysis even if you don't care about our stated reasons for not using them.*

1.5.1 False Starts

Before we start down our path to DRACL, we need to cover a few false starts just to rule out some design directions.

1.5.1.1 Bearer Credentials

Before going into real access control systems, we need to talk about bearer credentials. You've almost definitely run across them, usually in the form of "secret" links. That is, if you know (are a bearer of) the link, you can access the resource.

So, we could just give everyone secret links to the content they can access. Unfortunately, this really just side-steps the problem. We'd still need a system for managing them — i.e., distributing and revoking them. Furthermore, we'd need to make sure that users couldn't accidentally leak their credentials. It's surprisingly easy to leak a URL — e.g., by putting it in a document that gets indexed by google. At the end of the day, URLs just weren't designed to be credentials.

Some systems like Google's Macaroons [15] fix these problems by issuing short-lived, bearer credentials that can optionally require the bearer to prove knowledge of some additional key — separate from the macaroon. Macaroons even support delegation. However, because macaroons are short lived, we'd still need to design a system for issuing, reissuing when they expire, and managing Macaroons.

In short, bearer credentials are a useful component of access control systems but don't really solve the problem. As a matter of fact, DRACL ends up using a form of bearer credentials internally (we call them ACL Keys but more on that later).

1.5.1.2 Encrypt All The Things

Another solution is to just encrypt all content on content hosts such that only authorized consumers can decrypt the content.

This has *perfect* privacy. Nobody has to authenticate to anyone and no metadata is leaked — other than, potentially, some information about how many consumers can access a piece of content (because most crypto systems leak the cardinality of the set of keys that can decrypt something). Basically, producers just encrypt the content using all recipients' public keys (using some form of key-private encryption) and upload it anonymously to a content host. Consumers download it and try decrypting it.

This is great because it's simple. It doesn't require any infrastructure beyond a PKI. If a producer knows a consumer's public key, they can grant them access to a piece of content.

However, this would basically require reworking the entire web. Content hosts want to be able to integrate with the content they serve. If it's encrypted, they basically become file sharing services.

Additionally, this only supports static content out of the box. It's possible to build applications that operate on encrypted data-sets but it's really hard. Again, this just won't work with the web as it is.

Finally, there's no ability to recover from compromise. Attackers *will* (e.g., the NSA) download all the encrypted content on the web and wait for an associated key to be compromised. An access control system should require a consumer to authenticate *before* they can get their hands on any form of the content.

Just encrypting everything is appealing from a simplicity and privacy perspective but it's simply not practical. Note: This doesn't mean that users shouldn't encrypt their social interactions when possible, it just means that encryption is not sufficient.

1.5.2 Zeroth Attempt

A naive solution is to have content hosts ask producers directly if a consumer should be able to access some content. This obviously won't work because we can't expect the producer's computer to be online all the time. As a matter of fact, for all we know, the producer may never come back online — they could be dead.

Additionally, this solution has a serious privacy problem: the producer learns that the consumer has tried to access the content (refer to our privacy requirements for an explanation on why this is bad). We'd like to stick some service between the producer and the consumer to make this harder. At the end of the day, the producer could simply run this service themself but we feel that it's reasonable to assume that most producers won't have the resources to do this and, even if they did, privacy conscious consumers could refuse to use such a service.

1.5.3 First Real Attempt

The first reasonable design would be to have every producer pick an authentication provider (AP) and have their AP act their behalf to handle all access control decisions in real-time. Basically, when publishing content, producers would also assign a content ID (CID) to the content in question and tell the content host: when a consumer tries to access the content identified by CID, ask AP if they should be granted access. The user would then tell their AP which consumers and/or groups of consumers should be granted access to the content identified by a CID.

On the plus side, this is an extremely simple system and meets many of our requirements. From a privacy standpoint, neither consumers nor content hosts learn anything about producers' social networks. Additionally, consumer activity is revealed only to the producer's AP, not the producer. From a security standpoint access can be revoked without contacting individual content hosts and compromise is all recoverable as nobody stores any security-sensitive state. From a developer friendliness standpoint, this system is great because it's simple. Unfortunately, it still has has a some major drawbacks.

1.5.4 Don't Be A Weapon

So far, content hosts have to make network requests on behalf of clients (consumers) to servers (APs) specified by clients (producers). From a security standpoint, this is very bad as it opens up the content host to resource exhaustion attacks and makes it possible to use the content host in DDoS attacks. In general, it's a bad idea to perform network operations on behalf of an untrusted party[29] and/or wait for an untrusted party[27].

There's a simple, standard solution: have the consumer contact the AP on behalf of the content host. That is, the content host asks the consumer's browser to present a certificate from the AP certifying that the user should be able to access the content identified by the CID. The consumer's browser then authenticates to the AP, gets the certificate, and returns it to the content host. Offloading this work to the consumer has the additional benefit of reducing the load on the content host under normal conditions. As a matter of fact, most access control systems, including the Kerberos Consortium's User Managed Access (UMA) [9], operate this way.

1.5.5 Storing The ACLs With The Content

Unfortunately, there's another problem with this system as described: the access control server needs to keep track of every piece of content ever published. This is not a problem in terms of storage capacity (the list will at most be on the order of megabytes of raw data). However, it does require a lot of unnecessary book-keeping and database writes on the part of APs: they have to store something every time one of their producers publishes a piece of content and ensure that none of these records are ever lost.

Our solution was to distribute the load by storing the ACL on the content hosts themselves. That is, we replaced the CIDs with access control lists (ACLs) stored on the content host encrypted with a key known to the AP. Now, instead of asking the AP if a user should be granted access to a piece of content, the content host asks the AP if a user is a listed in an ACL (or is a member of a group listed in the ACL). We encrypt the ACLs to ensure that no party learns anything they shouldn't. The AP still needs to store the producer's social network (their friends list and group definitions) but this is likely going to be much smaller than the entire list of content they've ever published.

1.5.6 Don't Trust Third Parties

At this point, our system is looking pretty good but we still haven't addressed a key goal: no third parties with perfect security or uptime requirements. In the system as described so far, a compromise of an AP compromises the entire system (well, all producers that use that AP) because the AP makes all the authorization decisions. Ideally, we want some sort of black-box that only hands out authorizations to the correct consumers but having every producer ship a literal black-box to their APs is obviously impractical. However, there's a better way: cryptography.

The standard way to do this with cryptography would be to use certificates and public key cryptography. That is, every consumer has an public key and the producer signs this public key with a (time limited) statement that somehow describes what the consumer can access. Unfortunately, by nature of public key cryptography, the using the public key and certificate would uniquely identify the consumer to the content host violating our goal of anonymous consumers. So, we can't just do this with certificates if we want to meet this requirement.

A alternative solution using symmetric cryptography would be to give every consumer some set of secret keys based on what content they should be able to access. Unfortunately, symmetric keys never expire; that is a symmetric key that can decrypt something today will be able to decrypt it tomorrow. The only way to revoke a consumer's access to content would be to update the ACL to remove any mention of the symmetric key. This violates our goal of being able to efficiently revoke access/remove a user from a group without updating every single ACL mentioning that user.

As traditional cryptography doesn't appear to be enough, we turn to functional cryptography. Using functional cryptography, producers can give their APs virtual black boxes that, given user ID and a specially constructed ACL, can construct a certificate usable only by the given consumer — using some consumer-specific secret

— if, and only if, they are a member of the ACL. We'll call this black box the certificate granting box (CGBB). Unfortunately, this isn't without drawbacks: this crypto is *very* computationally expensive — on the order of a second of CPU time. This violates our requirement of not not performing expensive computation on third party servers.

1.5.7 Distribute The Load

A simple way to fix this is to give the CGBBs directly to consumers instead of some AP. This actually takes the AP out of the picture entirely so we're no longer relying on the AP to have perfect uptime (no AP) and it isn't running any expensive computations (doesn't even exist).

However, now we can't remove users from groups. Before, we could go to the AP and swap out an old CGBB for a new one with an updated group structure. We trust that the AP will do this because AP's aren't actively malicious. However, now that the producer gives the CGBB directly to consumers, they can't take them back. That is, they could go to the malicious consumer and ask "pretty please delete that CGBB I gave you" but the consumer has no reason to actually do so.

We can solve this by making the CGBBs expire after a period of time. That way, while a producer can't take a time-limited CGBB, it will simply stop working after a while.

But now we've just re-introduced a problem from the zeroth attempt: we can't expect the producer to ever be online. What happens when the producer goes offline permanently (e.g, dies) and one of the consumer's time-limited CGBBs expires? In the system so far, that consumer permanently loses access to the producer's content.

The solution this time is to bring the AP back in a limited form. Now, instead of giving the CGBB to the AP, we give the AP a CGBB factory that can make time-limited CGBBs. This way, even if the producer dies, the AP can continue producing these CGBBs in perpetuity. Fortunately, the crypto behind the CGBB factory is significantly less computationally expensive than the crypto behind using a CGBB itself.

This solution is far from perfect. Now that we are relying on timeouts, consumers will be able to continue to access content after they have been remove from the group until their CGBBs expire. This problem isn't related to the specific crypto we are using, it's just an inherent problem in taking the AP out of the loop. We can, and do, improve this situation slightly in chapter 2.

1.5.8 No Unrecoverable Compromises

An astute reader may have noticed that we haven't covered the requirement of no compromises so far. As described so far, a compromise of the producer, for example, is completely unrecoverable because the producer knows how to make all the keys. We assure you that DRACL meets this requirement but it's a bit complicated and didn't play much of a role in motivating DRACL's design. If you want to know more, checkout the system overview (chapter 2).

1.6 Conclusion

At this point, you should be familiar with why we're building DRACL, how we went about designing DRACL, and have a sense of how it works. From here, we move on to a deeper look at how DRACL actually works and analyze to what extent we actually meet our requirements.

Chapter 2

System Overview

This chapter gives a top-down overview of the DRACL protocol without going into any of the crypto or protocol specifics. It also assumes an existing identity system that can map identities (names) to public, authenticated keypairs. The specific PKI infrastructure is beyond the scope of the core DRACL protocol.

First, we give a lightspeed overview of the protocol. Then, in the following sections, we explain in further detail. This chapter is intended to provide enough information to understand and evaluate DRACL and therefore goes into more detail than the Design section (§ 1.5) above but is not a full specification for the DRACL protocol.

2.1 Lightspeed Overview

This section is a lightspeed overview. Think of it as SparkNotes™ for DRACL.

DRACL uses groups as the unit of access control. That is, producers put their friends in groups and specify which groups should be able to access which resources. We do this both for enhanced user experience (manually granting access to a many of consumers is tedious) and efficiency reasons (ACLs don't need to explicitly mention every user that has access). For technical reasons, producers are limited to 1000 groups. Unfortunately, this also limits the number of consumers a producer communicate with directly (one-on-one) because groups are the unit of access control. See the user friendliness section (§ 3.2) for an in-depth discussion.

When uploading a protected resource to a content host, the producer also uploads a signed and cryptographically opaque ACL along side it. The ACL internally specifies which of the producer's groups should be able to access the content, without revealing this information to the content host or consumers, and publicly lists the producer's public key and AP. ACLs contain no sensitive information and never expire.

When accessing a protected resource, the consumer must prove membership in the

ACL. They first download the resource's ACL and learn where to find the producer's AP. They then identify themself to the AP and fetch the necessary keys and certificates (which can be cached to limit communication with the AP). Then, using these keys and certificates, they prove membership in this ACL to the content host. This proof reveals only that the consumer has chosen to prove that they are a member of the ACL, nothing more. Specifically, it reveals nothing about the consumer's identity to the content host or the producer's social network to any party.

At any time, the producer can create, delete, or modify (add/remove members to/from) their groups. After updating a group, the producer uploads a new secret key for each of its friends to its AP but does not update any ACLs. After a user is removed from a group, they will lose access to any existing content when their keys expire. However, any content published after a consumer is removed from a group will never be accessible that consumer even if they have non-expired keys (we include a "minimum key version" in all ACLs).

Because all parties in DRACL make extensive use of cryptography, we need a way to handle the compromise of cryptographic secrets (keys). To do so, we have the AP certify all public keys in the system with short-lived certificates. If a user believes their keys to be compromised, they can revoke their keys by asking their AP to stop signing their keys. We also allow users to re-activate their accounts assuming that the underlying identity system provides a way to migrate from one identity to another.

2.2 Keys

First, we need to be able to talk about the different types of keys in this system. The actual system has even more keys but these a the only ones you need to know to understand DRACL:

Consumer/Producer Identity Key The consumer and producer both have asymmetric identity keys managed by the assumed identity system.

AP Key Every AP has a key for reasons explained later.

ACL Key A key issued by every producer to each of their consumers for authenticating against all their (the producer's) ACLs. You can think of these keys as the edges that describe the producer/consumer relationship graph (one edge per relationship).

Producer Secret All the secret information needed for a producer to make their ACL keys.

2.3 Authentication

When a consumer attempts to access protected content, the content host first sends the content's ACL to the consumer. The consumer then fetches any their ACL Keys from the producer's AP if not already cached and pre-verifies that they will be able to successfully authenticate against the ACL. If pre-verification succeeds, the consumer and content host run the authentication protocol to prove to the content host that the consumer should be able to access the protected content. This is diagrammed in figure fig. 2.1. The parts of this diagram that aren't strictly speaking part of the authentication protocol are marked with double parentheses and covered later.

Before running this protocol but after receiving the ACL, the consumer learns the identity of the producer (listed in the ACL) and how many groups they are in grant them access to the resource (this is simply an artifact of the crypto) but not which ones. At this point, the content host learns nothing.

After running this protocol, the content host learns precisely that the consumer has chosen to authenticate against an ACL.

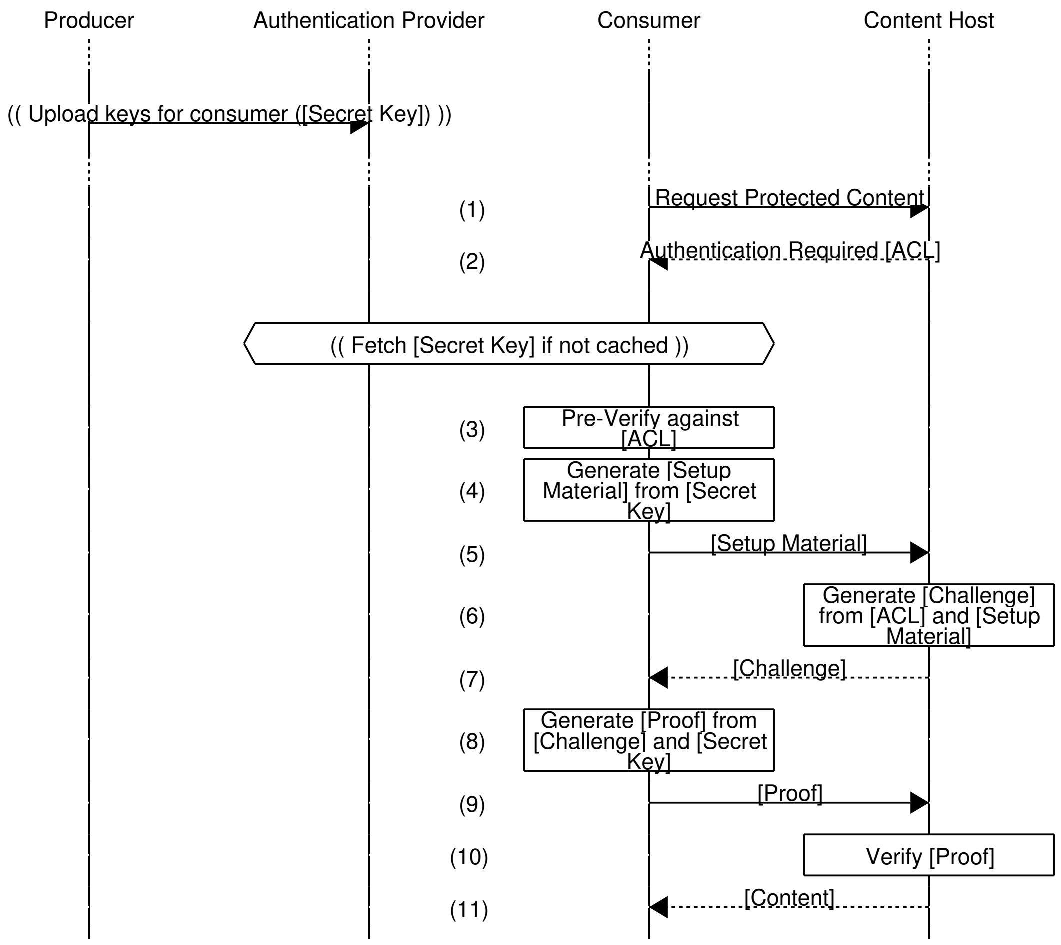
Producer
Authentication Provider
Consumer
Content Host
((Upload keys for consumer ([Secret Key])))
(1)
Request Protected Content
(2)
Authentication Required [ACL]
((Fetch [Secret Key] if not cached))
(3)
Pre-Verify against [ACL]
(4)
Generate [Setup Material] from [Secret Key]
(5)
[Setup Material]
(6)
Generate [Challenge] from [ACL] and [Setup Material]
(7)
[Challenge]
(8)
Generate [Proof] from [Challenge] and [Secret Key]
(9)
[Proof]
(10)
Verify [Proof]
(11)
[Content]

Figure 2.1: Authentication Protocol

2.3.1 The Content Host's Perspective

The previous section gives a top-down overview of the authentication protocol but doesn't make it clear what the content host actually does. To keep it simple, DRACL only exports two easy-to-understand functions to the content host: `is_acl` and `authenticate`.

`is_acl` simply verifies that an ACL is well formed. The content host should use this function to verify that ACLs uploaded by producers are valid (simply to avoid storing invalid ACLs).

`check_access` takes an ACL and a response from the consumer and spits out either `GRANT`, `DENY`, or `CONTINUE(reply)` with a reply to be sent to the consumer for further processing.

2.3.2 The Consumer's Perspective

When attempting to access a protected piece of content, the browser first asks the consumer if they want to prove that they have access and allows the consumer to remember this decision (for the specific content host, for the specific producer on this content host etc...). Additionally, if the consumer has never seen the AP listed in the ACL before, the browser asks the consumer if it trusts the AP. We do all this to prevent malicious parties from learning the consumer's identity (see § 3.1.2.4). To make this less of a pain, we expect browsers to ship with lists of trusted and untrusted APs and we tell the consumer if the ACL has been signed by a producer they know (i.e., they have previously shared/access content to/from them).

Finally, if the consumer's browser fails to generate the `[Proof]`, it warns the user. We do this to discourage content hosts from fingerprinting consumers (see § 3.1.2.1).

2.4 Key Distribution

In DRACL, consumers use secret keys (ACL Keys) given to them by producers when authenticating to content hosts. This means they need some way of getting these ACL Keys.

After choosing what groups each consumer should be in, the producer generates one ACL Key for each consumer encoding the groups to which the consumer belongs therein. The producer then encrypts the ACL Public Key with the consumer's identity key, signs it with its own identity key, and uploads the encrypted ACL Private Key, the associated ACL Public key, and the consumer's identity key to the AP.

To download the ACL Key, the consumer asks the producer's AP for either all ACL keys belonging to them (the consumer) or a subset thereof. When making this request, the consumer proves that their identity key hasn't been revoked using the identity system. Note, as the consumer identifies themself to the AP, the browser should ask the consumer if they trust the AP.

2.5 Account Compromise Recovery

DRACL supports account compromise recovery. When recovering a compromised account, a user is able to restore the security of their account in case their keys have been compromised. However, we don't currently support restoring publisher privacy after a compromise as this would necessarily require updating all old ACLs (see § 3.4 for details). There are two parts to account compromise recovery:

1. Account lockout.

2. Account restoration.

In this section, we assume that the underlying identity system has a way of marking identity keys as invalid/revoked. For example, the identity system could attach a short-lived certificate to every identity key.

To support account lockout, (1) all ACL keys issued by DRACL expire rapidly (on the order of hours) and (2) both the producer *and* the producer's AP must work together to make the ACL keys. This means that a compromise of either the producer or the AP doesn't compromise the entire system.

To prevent an attacker with a compromised but revoked producer identity key and from accessing that producer's content, we include a short-lived certificate signed by the producer's AP in the "setup material" of the authentication protocol (distributed along with the ACL Keys). If a producer notifies their AP that their keys have been compromised, the AP will stop producing this certificate effectively locking out all access to the producer's content.

To prevent an attacker with a compromised but revoked consumer identity key from accessing content from other producers, we have the AP stop issuing this certificate for the compromised consumer only. Importantly, the AP can do so without help from the producer because the producer may be dead for all we know.

To restore an account, we rely on the identity system. To support account restoration, the underlying identity system must provide a mechanism for securely transitioning from one (compromised) identity keypair to another.

However, after transitioning to a new key, users do not proactively replace old ACLs as this would require additional work and implementation complexity for content hosts. Instead, we continue to use the old ACLs as-is because our system's security does not rely on any non-ephemeral secrets *other* than the identity key.

Chapter 3

Assessment

Above all, we've designed DRACL to be implementable in practice, not only in theory. Additionally, DRACL is privacy preserving, secure, and developer and user friendly. Finally, as a decentralized system, DRACL promotes user-freedom. This section evaluates how we stack up against our requirements for a "good" access control system for the web.

3.1 Privacy Preserving

DRACL preserves the privacy of both producers and consumers. Preferably, DRACL would reveal nothing other than whether or not some user should be able to access some resource. Unfortunately, this goal is impossible to achieve given our efficiency constraints. However DRACL provides reasonable privacy given our performance constraints and some choices we've made when confronted with unavoidable privacy/security trade-offs.

Perfect privacy is impossible given our performance constraints because not doing something reveals that something hasn't happened. For example, because the producer does not update ACLs after removing consumers from groups, a malicious content host and consumer can collude to definitively learn that the consumer has been removed from a group (because the ACL hasn't been updated so the group structure must have been).

Below, we summarize what information DRACL reveals to the various parties involved:

Everyone Every ACL includes the producer's public key so the content host and anyone attempting to access a resource learns something about the identity of the producer (see § 3.1.1.2 for why). This doesn't mean they learn the "real" identity of the producer but they do learn an identity. Additionally, anyone can learn whether or not a producer has put them in at least one group.

Content Host Content hosts know the content (which could be an encrypted blob) and learn whether or not a given (anonymous) consumer chooses to prove that they have access to piece of content.

Consumers Consumers learn whether or not they can access any resource at any point in time. However, they do not learn *why* they have access. That is, they do not learn what groups they are in or what groups grant them access to any given resource. See section § 3.1.1.1 for some caveats and exceptions (specifically, they learn how many groups grant them access).

Authentication Providers Authentication Providers learn their users' social networks but can't act on behalf of their users or access their content.

Producer An honest but curious producer does not learn if and when specific friends access their content. Given sufficient collusion and effort, producers can learn something about who accesses what when. However, in practice, we don't believe that this will be much of an issue See section the on consumer privacy (§ 3.1.2) for a detailed discussion.

The following sections detail what privacy guarantees we provide and what information we leak.

3.1.1 Producer Privacy

In general, DRACL provably reveals (almost, read on) nothing about the producers social network to any third party other than the producer's AP. The following sections detail the exceptions.

3.1.1.1 Reasonable Privacy

We have three privacy leaks that aren't entirely a result of some performance or privacy trade-off.

First, consumers can learn how many of the groups they are in allow them to access a resource. That is, given the set of groups a consumer is in, $\mathbb{U}$, and the set of groups that can access a resource, $\mathbb{R}$, the consumer can learn the size of the intersection, $|\mathbb{U} \cap \mathbb{R}|$. This is simply an artifact of the cryptographic protocol we are using for authentication. We believe we can fix this by using stronger (more expensive) cryptographic primitives but haven't bothered exploring this avenue because they're too inefficient. Furthermore we believe this should be fixable without sacrificing performance but leave it to future work.

Second, anyone can learn whether or not an ACL has changed since the last time they accessed a piece of content. In theory, one could randomize challenges so that they look fresh every time. However this is likely more trouble than it is worth as, for security reasons, we have content hosts present a signed copy of the ACL along with challenges. Unfortunately, this does allow users to prove that they have been added to or removed from some (but not which) group with certainty by learning that they have gained/lost access to content without the ACL changing. However, regardless of what we do, consumers can learn this with high probability, but not with certainty, by simply assuming that gaining or loosing access to a set of content is more likely to be the result of being added or removed from a group than the result of each of those ACLs having been changed in a short period of time.

Third, consumers can determine a lower bound on when an ACL may have been crated because we record the "time" — technically a monotonic counter, not the actual time — the producer last removed a consumer from a group before creating an ACL in the ACL itself. This allows us to guarantee that all content published *after* a consumer is removed from a group is never accessible to that consumer.

3.1.1.2 The Producer Is Public

In DRACL, the producer's public key is publicly visible in the ACL itself. This may not be the producer's real identity, but it still identifies them. We had four options:

1. List the producer (what DRACL does).
2. Allow "friends" to learn the identity of the producer. By "friends" we mean authorized consumers of *some* content controlled by the producer.
3. Reveal the producer to the content host only.
4. Don't reveal the producer at all.

It is provably impossible to implement option 4 without sacrificing performance. Basically, because we don't update ACLs whenever a group definition changes some party must use some producer-specific information at some point in the authentication protocol. Because the AP can't participate directly in every access check, this party must be either the content host or the consumer. We've included a complete proof in § A.6.1

Implementing option 3 would either violate our security guarantees and open content hosts up to denial of service attacks or force the producer to contact every affected content host when their social network changes. As shown above, either the content host or the consumer needs to learn the producer-specific information. To prevent the consumer from learning the identity of the producer, we'd either have to have the content host fetch this information from the producer or have the producer send this information to the content host. The first option violates our rule that the content host must never make network requests on behalf of DRACL. As a matter of fact, this would be the worst case scenario: the content host would be making a network request to a server specified by a client (the producer) on behalf of a client (the consumer). On the other hand, having the producer send the producer-specific information to each content host ahead-of-time (whenever their social network changes) would complicate the content hosts (they would have to listen for updates from an external service) and force the producers to contact every content host they have ever used whenever they change the members of one of their groups.

If we were to implement option 2, DRACL would lose unidirectional friendships, scale horribly at the tail, or make it easier for a content host to deanonymize consumers.

Case 1: Consumers could remember a small list of producers with which they are actually friends. For each producer, they could store a producer-identifying secret that allows them to recognize an ACL as having been authored by this producer. However, that would make it harder to make new friends. In DRACL as designed, if a producer knows about a consumer, they can grant that consumer access to content. If consumers had to pick a short list of "friends" that could grant them access to content, we'd lose this property.

Case 2: Consumers could keep track of every producer who has ever shared a piece of content with them. Unfortunately, this would require them to possibly keep track of many producers and require an infrastructure for notifying consumers when some producer has shared some content with them. Additionally, for popular users, this list could grow very large. Worse, the size of this list scales based on actions taken by *other* producers, not the consumer in question. Finally, to actually authenticate against an ACL, consumers would have to linearly search through the set of producer-identifying secrets to find the producer that authored the ACL. We sketch a proof for this statement (making a reasonable assumption that we don't bother proving) in § A.6.2.

Case 3: Consumers could ask the AP (listed in the ACL) for the producer-identifying secret when they encounter an ACL with an unknown producer. They could also store a small set of these (the one's associated with close friends). However, if the AP just gave these keys out to anyone who asked, we'd gain no privacy. Therefore, the consumer has to identify to the AP to allow the AP to determine if the consumer is a friend of the producer in question). Unfortunately, at this point, the producer has little reason to trust the AP with their identity as they don't know what producer created the ACL. As a matter of fact, the content host could have created the ACL in an attempt to get the consumer to identify to an AP run by the content host. Details of this specific attack are discussed in § 3.1.2.4.

In short, we make the producer public because it allows us to achieve better usability and performance and allows the consumer to make an informed decision when identifying themself to a third party.

3.1.1.3 The AP Learns The Social Network

In DRACL, APs learn the social networks of their producers. Specifically, for any given producer, they learn which consumers that producer has put in at least one group. However, they do not learn the group assignments.

This is a consequence of how we allow APs to revoke a consumer's access to a producer's content without involving that producer. Basically, the AP needs to be able to verify that the consumer's key has not been revoked when the consumer fetches their keys; this identifies the consumer to the AP. Furthermore, remember that the AP issues its own keys to consumers on behalf of its producers. It needs to know for which producer it's issuing these keys so it can properly sign them. This means that AP knows for and to whom it's issuing the keys so it knows that the producer has put that consumer in a group.

One solution would be to have the producer's AP send consumers' keys to some trusted broker (i.e., something like an AP but for consumers) and allow the broker to know who the consumers are but not the producer's AP. This way, one party (the broker) knows the consumer but not the producer, and the other party (the AP) knows the producer but not the consumer. Unfortunately, this would require a lot of up-front work and network traffic, even if the consumer never views content published by some producer. By a lot, we mean that every time an ACL key expires (on the order of hours) the producer's AP would have to send out a 30KiB key per consumer/producer relationship. Individually, this isn't much data. However, assuming there are two equally popular APs, ACL keys expire twice per day, and all producers have shared with 300 consumers on average over the course of their existence, each AP would have to send 4MiB per producer to the other AP twice a day. That's not that much data for active producers/consumers but these data would have to be sent for inactive producers/consumers as well, twice a day. On the scale of Facebook, that's 8PiB (that's *Petabytes*) per day.

However, this could be solved without adding a new party with some fancy crypto and an anonymizing network like Tor. Basically (enormous simplification), the producer would have their AP sign the consumer's ACL Keys with the consumer's identity cryptographically hidden and then re-upload them to the AP anonymously with the their (the producer's) identity cryptographically hidden.[1]

Unfortunately, even this simplified scheme is complicated. It would require a some fancy crypto (something like a partially blind partially homomorphic signature algorithm), would require multiple rounds of communication between the producer and their AP (some of them over anonymizing networks), and would still be vulnerable to timing correlation attacks. By timing correlation attack, we mean that, given that producer A uploaded a key for signing at time T and an anonymous producer

[1]In slightly more detail, the producer would get the AP to sign a consumer's ACL Keys where any information identifying the consumer has been cryptographically hidden (blinded) but the parts identifying the producer are visible (to the AP). Then, the producer would unblind the parts mentioning the consumer and blind the parts mentioning the producer and re-upload the altered key to the AP over an anonymized connection (e.g., over Tor). The consumer could now safely authenticate when downloading the keys because they wouldn't be tied to the producer in any way (that part has been cryptographically hidden). The consumer would then have to unblind the part that mentions the producer to be able to successfully authenticate to content hosts.

uploaded a key for consumer B at time $T + \epsilon$, A is probably a friend of B.

Basically, a perfect system to hide the producer's social networks from APs but such a system would be significantly more complicated.

3.1.1.4 Recovery From Compromise

If the producer's keys are compromised, their entire social network is leaked. There's obviously nothing we can do about this. This is similar to a Facebook account being hacked.

However, a problem specific to DRACL is that, after a producer's account has been locked down from a security perspective, an attacker can still learn which groups are members of (old) ACLs. This is an artifact of the fact that we do not go back and replace old ACLs on account compromise. If the producer feels so inclined, they could manually go back to every content host and replace their old ACLs but we're not going to bake that feature into DRACL. Basically, we feel that, at this point, the chicken has flown the coop so to speak.

3.1.2 Consumer Privacy

DRACL allows consumers to anonymously access content. Specifically, DRACL attempts to prevent content hosts and untrusted third parties from tracking consumer behavior and browsing habits. Furthermore, it avoids notifying the producer directly when a consumer "sees" a piece of content.

However, there are ways in which malicious parties can try to use DRACL to identify and/or track consumers in some cases. Below we first discuss passive attacks that can let a content host learn something about the identity of a consumer and then finish with an active attack and some mitigations.

3.1.2.1 Fingerprinting

One problem with any access control scheme is that the content host can fingerprint consumers based on what they can and cannot access. The consumer could make every access request look like it's coming from a new client but this is infeasible in practice. Instead DRACL allows the user to decide if and when to authenticate and warns users (loudly) when authentication fails. This way, consumers control what information they give to content hosts and content hosts can only confirm answers they already suspect. While there are other ways for content hosts to identify users (i.e., they could be logged in) DRACL should not leak this information.

3.1.2.2 Timing and Caching

For increased performance and reliability, consumers cache keys retrieved from producers. Unfortunately, caching tends to leak timing information. In our case, we were worried that a curious content host could learn whether or not a consumer “knew” a producer even if the consumer chose not to authenticate based on whether or not the consumer had already cached the producer’s key.

To mitigate this this, consumers pre-verify (§ 2.3) if they will be able to access a piece of content. If this pre-verification fails, they never even initiate the authentication protocol. This means that the content host can only possibly learn timing information if the user has access to the content in question and has chosen to access it. However, in this case, the producer already learns that the consumer “knows” the producer.

3.1.2.3 Side Channels

Content hosts will likely learn about the consumer's browser, IP address, etc. This is beyond the scope of DRACL system and users that require true anonymity should use the Tor Browser[8] or Tails[7].

3.1.2.4 Directly De-anonymizing Consumers

Unfortunately, consumers who habitually click "yes" for everything can be trivially de-anonymized by a malicious content host. Here we discuss this problem and our mitigations.

When fetching their keys from an AP, consumers must identify themselves to properly handle consumer key revocation (see § 2.5). Unfortunately, this means that an attacker (not necessarily a friend of the consumer!) can post a resource to a content host along with an ACL that lists a unique AP domain controlled by the attacker. If a consumer chooses to access this content, they'll identify themselves (using the identity system) to this snoopy AP. Because the domain is unique to the resource, the attacker learns that a specific consumer has accessed a specific resource. This is obviously very bad.

To mitigate this, consumers only talks to trusted APs (see § 2.3.2). Furthermore, because ACLs are signed by producers, consumers can choose simply not authenticate against ACLs signed by producers they don't recognize or trust. While a malicious producer could deanonymize a user by tricking them into into using a malicious AP, we hope that consumers in DRACL will have a better taste in friends.

There are a few alternatives:

1. Don't have the consumer identify to the producer's AP. Unfortunately, this means that the producer's AP can't verify that the consumer's key hasn't been revoked. We could provide an option for a consumer to claim: "I know what I'm doing, I'll never lose control of my key." However, such "expert only" options can be dangerous in practice so we would have to consider this very option carefully before adding it.
2. Push keys to the consumer's AP and have the consumer fetch them from there. This approach has already been discussed and refuted in section § 3.1.1.3.
3. Have consumers keep track of "active" friends and fetch ACL keys from them preemptively. This approach has already been discussed and refuted in section § 3.1.1.2.

So, malicious APs are a problem and consumers should avoid authenticating ACLs from untrustworthy sources. At the end of the day, this is similar to not logging into shady websites with something like Facebook Connect.

3.2 User Friendly

We have designed DRACL (at the system level) to be user friendly and unobtrusive. We meet our requirements but there's still room for improvement.

3.2.1 What We Got Right

DRACL significantly reduces the cognitive burden on users and allows them to be social on the web without performing tedious repetitive tasks.

Users only have to remember one set of credentials (their DRACL credentials). DR-ACL can even be used to sign in to other systems by treating accounts as resources. This, incidentally, also allows multiple people to share a single "account" without sharing passwords. We could probably go on-and-on about how useful this is — shared credentials are a big problem in the corporate world — but that's beyond the scope of this document.

Users can take their their social network with them from content host to content host without giving up privacy.

Day to day, DRACL should be unobtrusive and integrate well with content hosts. DRACL is designed to be implemented in the browser itself and to integrate deeply with the browser. Unlike systems like OpenID, Facebook Connect, and Google Sign-In, it won't have to open up a "sign-in" tab whenever the user wants to authenticate, it can simply display a dialog box. Furthermore, on content hosts a consumers uses regularly, when accessing content from producers that the consumer "knows", the browser doesn't even have to ask the user to authenticate. It can just silently authenticate on their behalf and display the content (if instructed to do so by the consumer).

These last two issues have been cited [12] by Mozilla as reasons reasons their authentication system, Persona, failed.

3.2.2 Where We Could Improve

DRACL has some significant usability drawbacks: it's slow, revocation isn't instantaneous, and the maximum number of consumers to which a producer can grant access to content is limited (as it is on Facebook).

The client-side DRACL operations are *very slow.* They take about 1-2 seconds on my 6 year old laptop so they should take at most a second on a recent laptop. Worse, if a consumer has not accessed a piece of content from a producer recently, they have to make a second round trip to that producer's AP to fetch their keys. However, when compared to the time and effort it takes a user to sign in to a website, this is actually quite cheep.

Revocation isn't instantaneous. This was discussed from a security standpoint in section § 3.4 but it's really more of a usability problem. The UI will have to make this clear to producers and warn them to manually re-create ACLs for any content that they need to lock-down immediately. While unfortunate from a usability standpoint, this problem is unavoidable given our requirements; instantaneous revocation would require the AP to participate directly in the authentication protocol.

Due to the crypto primitive we use, we have to limit the maximum number of groups a producer can make. Unfortunately, our unit of access control is a group; to share with an individual, you have to put them into an personal user group. This means that the limit is effectively $g = |groups| + |consumers|$. Currently, we've set this limit to 1000. We could increase this limit but DRACL's keys, ACLs, and authentication protocol all scale linearly with this constant. I get the following numbers on my laptop:

Consumer Authentication Time	$(.0015g + \epsilon)\,\mathrm{s}$	$\approx 1.5 \pm .5\,\mathrm{s}$
Acl Size	$(128g + \epsilon)\,\mathrm{B}$	$\approx 128\,\mathrm{KiB}$

where $g = 1000$

We believe these numbers can be improved simply by improving the underlying cryptographic protocol. However, we leave that for future work.

Also, we don't currently provide a way to delete existing groups. However, implementing this would be a straightforward improvement so we leave it to future work. Basically, as "dead" groups accumulate, one would occasionally transition to a new ACL Key (without the dead groups) and use the new key for new content. This does mean that consumers would need every ACL Key ever created by a consumer but, in practice, there shouldn't be that many. This is basically a simple mark-compact garbage collection mechanism.

Overall, DRACL provides a large usability benefit over existing systems but our security and performance requirements do impose a significant usability cost.

3.3 User Freedom

DRACL promotes user freedom by being decentralized, by allowing consumers to freely move between content hosts, and by enabling developers to easily create new services.

Simply by being decentralized, DRACL promotes user freedom by allowing producers to choose their authentication provider. This ensures that no authentication can become a monopoly.

By allowing producers to manage their social network through a single authentication provider, DRACL allows users to carry their social network with them from content host to content host. This reduces the barrier to moving between content hosts and therefore promotes user freedom in their choice of content host.

Finally, by allowing producers to share content with consumers on any content host even if the consumer doesn't have an account at that content host, we've reduced the difficulty of bootstrapping a new content host. This should make it significantly easier for companies to break into the social network market which should increase the number of social networks users can choose from.

One key missing component from our current design is the ability to move between authentication providers. That is, we provide no simple way to migrate off one AP to another. Adding this to DRACL should be relatively straightforward but it would complicate an already complicated system.

Overall, DRACL promotes user freedom as much as possible while still remaining practical.

3.4 Secure

DRACL is secure in practice. Being secure in some perfect world where all software is perfect and bug free isn't practical. To this end, DRACL doesn't allow unauthorized access to content (no exceptions), supports recovery from account compromise, supports efficient revocation, and avoids being a vector for denial of service attacks. DRACL's security is rooted in a standard CA system so all *security* compromises are recoverable as long as the CA system works. This obviously isn't perfect because the CA system is deeply flawed but this can always be replaced if and when a better alternative becomes available.

Only authorized users, the producer, and the content host are able to access resources protected by DRACL. Specifically, DRACL doesn't allow the AP to impersonate its users and access their content.

However, a malicious AP can time travel. That is, a malicious AP can revert any consumer's view of any producer's social network to any previously valid state. They can do this because the producer's ACL keys never expire (so that the producer doesn't have to keep coming back online to re-sign them). In practice, this means that a malicious AP can prevent a consumer from accessing content they should be able to access and allow a consumer to access content that would have been accessible to them in a previous incarnation of the producer's social network in the content's current form (the AP can time-travel the social network, not content). A security conscious producer can put an expiration date on the ACL keys they issue to limit the effectiveness of such an attack but this would force them to re-issue their ACL

Keys before they expire to avoid locking out access to *all* their content.

Producers can efficiently revoke access to content. To facilitate this, DRACL allows users to be removed from groups without updating every ACL mentioning the now modified group. Much of the complexity of DRACL stems from this requirement. Unfortunately, in order to do this efficiently, we sacrificed some security. Specifically, content published by a producer before they remove a consumer from a group will remain accessible to that consumer until their producer-specific certificate expires. This is a more limited form of time-travel. We mitigate this by making producer-specific certificates short-lived — they expire on the order of hours.

However, there is an important limitation on time travel in DRACL: consumers with a time-traveled view of the social network — either due to a malicious AP or recently revoked access — cannot access content published in later incarnations of the social network. This is due to an epoch counter we include in every ACL and every ACL Key. Basically, an ACL Key's epoch must be greater than the ACL's epoch. A lower epoch indicates that the key is from an old view of the social network and the content host will deny access.

Both consumers and producers are able recover and secure compromised accounts.

We achieve this by having APs act as semi-trusted third parties that certify their user's keys with short-term certificates. Once an AP learns that one of its users has been compromised, it stops signing their keys. The AP then works with the user to transition the user to a new set of keys (authenticating the user's identity out-of-band). For a quick overview of this protocol, see section § 2.3.

Unfortunately, DRACL does not support restoring a producer's privacy after an account compromise. That is, after an attacker obtains a set of "producer secrets", they can determine what groups can access any given resource and, if a consumer colludes with them, what groups that consumer is in.

DRACL stores *no* sensitive information on the content host. This means that, if a content host is compromised, they can simply restore from backup, update their software, and continue on as if nothing happened — as far as DRACL is concerned at least.

The long-term security of DRACL rests entirely on the identity keys and the CA system. To fully — time travel doesn't count — break the security of DRACL, an attacker would have to obtain both the producer's identity key and a certificate for AP's domain. The "producer secrets" are actually only secret because they describe the producer's social network, not because they must remain secret for security reasons.

Finally, DRACL cannot interfere with the operation of content hosts. To achieve this, we guarantee that all DRACL operations performed by the content host will complete in constant time. As a direct consequence of this, contents host never initiate network requests on behalf of DRACL.

3.5 Developer Friendly

To be developer friendly, DRACL wraps all content-host logic in a simple (interface wise), environment-agnostic library that should be easy to integrate into existing content hosts. As noted in the system overview, we export three functions: `is_acl`, `make_challenge`, and `validate_response`.

First, these functions are dead simple. We expose no functionality beyond simple access control checks. While more a complex interface may provide slightly better efficiency through, e.g., caching, this would make it easier to misuse these functions.

In addition to being simple, these functions perform *no* network requests and run in constant time (as noted in the security § 3.4 section above). This significantly reduces content host implementation complexity because it means that these functions can be called synchronously.

Finally, access control checks never lead to database writes. This is important for high-performance applications where reads are often performed out of read-only caches.

However, DRACL isn't perfect from a developer's perspective. ACLs themselves are approximately 128KiB, mostly due to our expensive crypto. This means that developers will have to store a 128KiB binary blob somewhere. Furthermore, verifying that a consumer has access to a piece of content requires that the content host perform some reasonably expensive crypto. This crypto takes at most a few milliseconds on modern hardware but that's still significantly more expensive than, e.g., accessing an in-memory database.

3.6 Scalable

DRACL scales to the entire web. To achieve scalability, we put as little load on the AP as possible.

APs aren't required to have perfect uptime. That is, in a steady state, consumers can continue to access content and producers can continue to publish content without having to contact the AP for every operation.

APs have storage and computation loads proportional to the number of consumer-producer relationships, not proportional to the amount of content published by a producer. Therefore, the cost of running a DRACL AP should scale approximately proportionally to the number of users of the service. This means that, in a steady state, the cost of running an AP should actually decrease over time as storage, bandwidth, and CPU power become cheaper.

DRACL is decentralized. This give instant scalability as the number of APs can scale with demand.

However, it is unfortunate that APs need to do some crypto in real time when consumers fetch their ACL keys. Currently, APs need to do some crypto when handing out ACL Keys to consumers in order to verify that the consumer's identity key hasn't been revoked. However, we could do all the crypto offline ahead-of-time using some fancier crypto schemes. Unfortunately, this gets a bit complicated and may not be worth doing in practice. Regardless, this is worth exploring as it would allow the AP to choose when and where to perform moderately expensive calculations which could significantly reduce the cost of running an AP when compared to performing these calculations on-demand.

3.7 Future Work

Throughout this chapter, we've identified future work that could improve DRACL. Here, we list them off for easy picking.

- Don't leak the number of groups that grant a user access to a resource.
- Look into allowing APs to perform all crypto off-line instead of on-demand.
- Provide a way to garbage collect empty groups. To do this, producer would literally just transition to a new set of Producer Secrets (see § 2.2) without defining the dead groups in the new secrets. However, there are some nuances that don't make this entirely trivial.

- Cut out a round trip between the content host and the consumer. Currently, the consumer needs to send the content host some information before the content host can generate a challenge. This is actually only necessary for consumer anonymity, not security or producer privacy (see the anonymity proof). However it really should be possible to do this without the extra round; it's just non-obvious.
- Increase the 1000 consumer/group limit and/or reduce our size/time complexity (§ 3.2.2).
  - We believe that we could double this limit simply by improving the proof of the underlying crypto — it doubles the size of the keys to make the proof go through but this may not be necessary.
  - We have explored the possibility of making it possible to share with some number of groups and some (small) number of consumers directly. This way, we wouldn't have include the number of consumers in the 1000 group limit. However, this would require modifying the underlying cryptographic protocol and is non-trivial. By non-trivial, we mean that we don't know exactly how to do it but believe it to be possible. We could, with relative ease, allow granting access to some number of groups or (exclusive) an single consumer (identified by a unique ID) but this pro-

vides dubious benefit and complicates the underlying crypto (and may introduce some privacy issues, we haven't fully explored this option).

– We could provide some sort of hybrid scheme where producers have multiple sets of groups — let's call them classes — each with a 1000 group limit. Consumers would be able to learn which classes they are a member of (e.g., "MIT Student", "Class of 2014", "US Citizen") but nothing about the actual group structure within the classes.

Chapter 4

Authentication Protocol

This chapter covers the authentication protocol in detail from a systems perspective without explaining the underlying crypto.

4.1 Primitives

Before we can cover the authentication protocol, we need to know what tools we have in our arsenal. We won't explain how they work quite yet but we need to know what we can use.

4.1.1 Standard Tools

In DRACL, we use standard tools like HMAC, hashing, symmetric encryption, and asymmetric encryption. If you are unfamiliar with any of these terms, please go to Wikipedia and familiarize yourself with them now. Otherwise, you will be very confused.

Cryptographers: This chapter is targeted at non-cryptographers. Please make reasonable assumptions where appropriate. When we say encryption, we mean authenticated randomized encryption. When we say hashing, we're talking about cryptographically secure hashing with preimage resistance.

4.1.2 Zero Knowledge Set Intersection

DRACL needs a way for consumers to be able to prove to the content host that the set of groups that they are in intersect with a set of groups listed in an ACL (without revealing either set of groups to either party). Unfortunately, this didn't exist so we had to create a new primitive based on something called inner product encryption (see chapter 5).

An admin (in our case the producer) can generate special encrypted verifier sets ($\texttt{Verifier}(\mathbb{A})$) and encrypted prover set key-pairs ($\texttt{ProverPublic}(\mathbb{B})$, $\texttt{ProverPrivate}(\mathbb{B})$). Someone with one of these verifier sets can generate encrypted challenge sets ($\texttt{Challenge}(s_1, \mathbb{B})$ where s is a secret). Someone else with a prover set can prove that their prover set intersects with a given challenge set by producing $\texttt{Proof}(\texttt{Challenge}(s_1, \mathbb{B}), \texttt{ProverPublic}(\mathbb{B}))$ without knowing s. We use this to allow consumers — with prover sets that describe the groups that they are in — to prove to content hosts — with challenge sets that describe the groups that are in the ACL — that the consumer is in some group listed in the ACL. All of these functions take an implied parameter p that describes the producer's environment (i.e., their secret keys).

Note: Crucially, all these sets are encrypted and hidden from all parties except the admin (producer).

4.2 Protocol

First, we cover the protocol without explaining the cryptography behind it. We've also duplicated the protocol diagram from § 2.3 (fig. 4.1) and refer to it liberally.

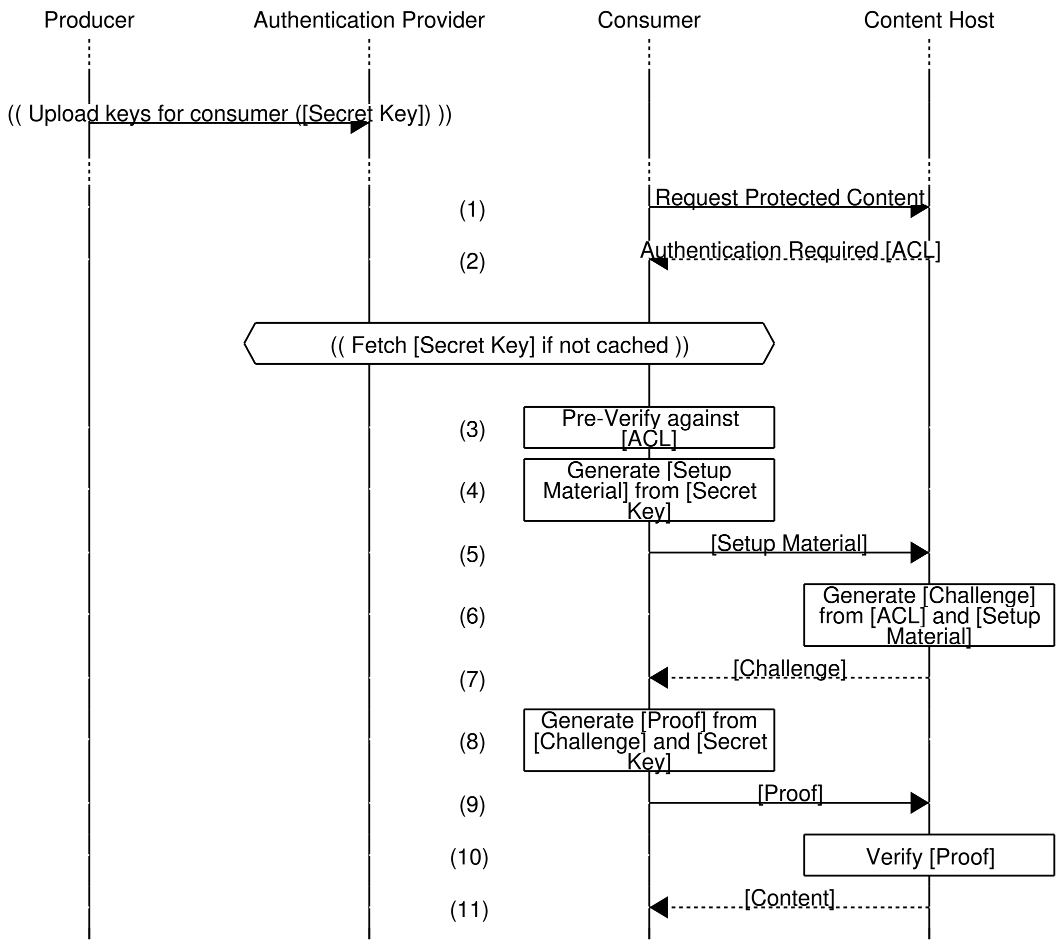
Producer
Authentication Provider
Consumer
Content Host
((Upload keys for consumer ([Secret Key])))
(1)
Request Protected Content
(2)
Authentication Required [ACL]
((Fetch [Secret Key] if not cached))
(3)
Pre-Verify against [ACL]
(4)
Generate [Setup Material] from [Secret Key]
(5)
[Setup Material]
(6)
Generate [Challenge] from [ACL] and [Setup Material]
(7)
[Challenge]
(8)
Generate [Proof] from [Challenge] and [Secret Key]
(9)
[Proof]
(10)
Verify [Proof]
(11)
[Content]

Figure 4.1: Authentication Protocol

4.2.1 Setup

4.2.1.1 ACL

In DRACL, every ACL includes a `Verifier`($\mathbb{A}$) where $\mathbb{A}$ is a set that describes the groups that should be granted access. Additionally, every ACL lists the producer's public identity key and the producer's AP. Finally, every ACL is signed by the producer's identity key to allow the consumer to verify that it hasn't been tampered with.

4.2.1.2 ACL Key

Every consumer's ACL Private Key includes a `ProverPrivate`($\mathbb{B}$) and their ACL Public Key includes a `ProverPublic`($\mathbb{B}$) where $\mathbb{B}$ encodes the groups in which the consumer is a member. Additionally, the consumer has two certificates — one from the producer and one from the AP — on `ProverPublic`($\mathbb{B}$) that certify the time period for which it is valid.

4.2.2 Protocol

Note: *In our first pass at the protocol, we leave off a crucial privacy feature. Implementors must read this entire chapter.*

Abstractly, the protocol is simple: the content host chooses a secret scalar s, gives the consumer some encrypted form of s, and then gets back some form of s that the consumer wouldn't have been able to compute if the intersection of $\mathbb{A}$ and $\mathbb{B}$ were zero.

The numbers in this section refer to steps in the authentication protocol (fig. 4.1).

4.2.2.1 Attempt Access

When a consumer attempts to access a protected piece of content (1), the content host returns the signed ACL to the consumer which includes $\texttt{Verifier}(\mathbb{A})$ (2).

4.2.2.2 Pre-Verify

The consumer first verifies the signature on the ACL and decides whether or not to even attempt to authenticate against the ACL (e.g., they may not be friends with the producer).

They (the consumer) then pre-verifies (3) that they should be able to authenticate against the ACL by running the cryptographic part of authentication protocol locally (pretending to be the content host).

4.2.2.3 Generate Setup Material

Note: *Make sure to read § 4.2.3 for a crucial change to this step.*

If they choose to continue, the consumer sends their ACL Public Key to the content host along with the certificate (5).

The content host verifies the certificate on `ProverPublic`($\mathbb{B}$) and verifies that it is currently valid: the key hasn't expired and has been signed by both the producer and the producer's AP.

4.2.2.4 Challenge Generation

The content host then generates two secrets (s_1 and s_2) and creates a challenge (6):

$$\texttt{Challenge}(s_1, \mathbb{B}) = \texttt{MakeChallenge}(s_1, \texttt{ProverPublic}(\mathbb{B}))$$

They also use the consumer's ACL Public Key ($\texttt{ProverPublic}(\mathbb{B})$), s, and $\texttt{Verifier}(\mathbb{A})$ to predict the proof that the consumer will generate:

$$\begin{aligned} P &= \texttt{Proof}(\texttt{Challenge}(s_1, \mathbb{B}), \texttt{ProverPublic}(\mathbb{B})) \\ &= \texttt{PredictProof}(\texttt{Challenge}(s_1, \mathbb{B}), s, \texttt{ProverPublic}(\mathbb{B})) \end{aligned}$$

The content host can generate this without knowing an appropriate $\texttt{ProverPrivate}(\mathbb{B})$ because they know s_1.

The content host then computes $T = \texttt{Hash}(P)$, and symmetrically encrypts the (s_1, s_2) pair using T as the secret key to produce E (6).

The content host then sends $\texttt{Challenge}(s_1, \mathbb{B})$ and E to the consumer (7).

4.2.2.5 Proving Access

The consumer computes the proof from $\texttt{Challenge}(s_1, \mathbb{B})$, $\texttt{ProverPrivate}(\mathbb{B})$, and $\texttt{Verifier}(\mathbb{A})$:

$$\begin{aligned} P &= \texttt{Proof}(\texttt{Challenge}(s_1, \mathbb{B}), \texttt{ProverPublic}(\mathbb{B})) \\ &= \texttt{MakeProof}(\texttt{Verifier}(\mathbb{A}), \texttt{Challenge}(s_1, \mathbb{B}), \texttt{ProverPrivate}(\mathbb{B})) \end{aligned}$$

They then hash it to recover T, and then decrypt E to recover (s_1, s_2).

Now, the consumer re-creates $\texttt{Challenge}(s_1, \mathbb{B})$ from $\texttt{Verifier}(\mathbb{A})$ using s_1 to verify that the content host followed the protocol correctly. This ensures that the content host can't fish for additional information by giving the consumer a malformed challenge or one that doesn't correspond to the ACL presented in § 4.2.2.1.

The consumer then symmetrically signs (HMACs) a statement "proof of membership for `domain_of_content_host`" with s_2 (8) and sends this back to the content host as proof that they should be granted access to the content.

4.2.2.6 Verification

Finally, the content host verifies the HMAC using s_2 (10), verifies that the consumer knows what content host they are talking to to prevent man in the middle attacks, and finally grants them access to the content (11).

4.2.3 Randomizable Public Keys

The protocol as described so far does not preserve consumer anonymity because `ProverPublic`($\mathbb{B}$) is unique to the consumer. However, we can fix this by giving consumers special re-randomizable keys. That is, we give consumers `ProverGenerator`($\mathbb{B}$) and a function `Randomize` that takes a randomization parameter r, the `ProverGenerator`, and spits out a randomized `ProverPublic`($\mathbb{B}$) and `ProverPrivate`($\mathbb{B}$).

Unfortunately, this means that we can't certify `ProverPublic`($\mathbb{B}$) directly using standard asymmetric cryptography because it's generated by the consumer. To fix this, we use a special malleable signature scheme to sign `ProverGenerator`($\mathbb{B}$) that can tolerate this re-randomization and put the public key of this scheme in the certificate.

We'll define $\texttt{HSigGenerator}(\texttt{HSigPublic}, G)$ to be a re-randomizable signature on a the $\texttt{ProverGenerator}$ G belonging to some key $\texttt{HSigPublic}$. We also modify $\texttt{Randomize}$ to accept a set of $\texttt{HSigGenerator}(G)$'s and spit out a matching set of $\texttt{HSig}(\texttt{ProverPublic}(\mathbb{B}))$'s. That is, given $G = \texttt{ProverGenerator}(\mathbb{B})$:

$$\begin{aligned}
&(\\
&\quad \texttt{ProverPublic}(\mathbb{B})_r,\\
&\quad \texttt{ProverPrivate}(\mathbb{B})_r,\\
&\quad \texttt{HSig}(\texttt{HSigPublic}, \texttt{ProverPublic}(\mathbb{B})_r),\\
&\quad \ldots\\
&) = \texttt{Randomize}(G,\ \texttt{HSigGenerator}(\texttt{HSigPublic}, G), \ldots)
\end{aligned}$$

We define a function $\texttt{VerifyHSig}$ that can verify some signature $\sigma = \texttt{HSig}(\texttt{HSigPublic}, \texttt{ProverPublic}(\mathbb{B}))$:

$$\texttt{VerifyHSig}(\texttt{HSigPublic}, \sigma, \texttt{ProverPublic}(\mathbb{B}))$$

Concretely, we make the following modifications.

4.2.3.1 Setup

We now give the consumer $G = \texttt{ProverGenerator}(\mathbb{B})$ in the setup phase (§ 4.2.1.2) instead of directly giving them $\texttt{ProverPublic}(\mathbb{B})$ and $\texttt{ProverPrivate}(\mathbb{B})$. Additionally, instead of directly certifying $\texttt{ProverPublic}(\mathbb{B})$, both the content host and the AP create their own malleable keypairs (consistent between all consumers), sign G with these key-pairs ($\gamma_p = \texttt{HSigGenerator}(\texttt{HSigPublic}_p, G)$), and then certify the key-pairs themselves.

4.2.3.2 Generate Setup Material

In the Generate Setup Material (§ 4.2.2.3) phase, we have the consumer generate fresh a fresh $\texttt{ProverPublic}(\mathbb{B})$, $\texttt{ProverPrivate}(\mathbb{B})$ and malleable signatures:

$$\begin{aligned}
&(\\
&\quad \texttt{ProverPublic}(\mathbb{B}),\ \texttt{ProverPrivate}(\mathbb{B}),\\
&\quad \sigma_{producer},\ \sigma_{AP}\\
&) = \texttt{Randomize}(r, G, \gamma_{producer}, \gamma_{AP})
\end{aligned}$$

4.2.3.3 Challenge Generation

In the challenge generation phase, we have the content host verify the consumer's $\texttt{ProverPublic}(\mathbb{B})$ by calling $\texttt{VerifyHSig}(\texttt{HSigPublic}_p, \sigma_p, \texttt{ProverPublic}(\mathbb{B}))$ and then verifying the certificate on $\texttt{HSigPublic}_p$ for both the producer's AP and the producer.

Chapter 5

Crypto

In the previous chapter, we assumed that there was some way for consumers to prove that their `ProverPrivate`($\mathbb{B}$) intersects with the ACL's `Verifier`($\mathbb{A}$). In this chapter, we explain how we actually do this. However, we first re-describe the entire protocol from scratch from a crypto perspective as opposed to a systems security perspective as we did above. This allows us to separate systems security concerns from cryptographic concerns and makes it easier to understand the crypto.

To reiterate the problem, we needed a way for consumers to prove that the set of groups in which they are a member intersects the set of groups listed in an ACL.

There are three requirements that make this difficult:

1. These sets must remain secret from both parties. That is, we allow neither consumers nor content hosts to learn what groups the consumer is in nor what groups have access to any given resource.
2. The consumer needs to actually be able to prove to the content host that the vectors intersect *without* identifying themself to the content host and/or granting the content host the ability to impersonate the consumer.
3. Consumers must lose the ability to prove membership after a period of time. That is, ACL Keys must expire.

We now describe some cryptographic tools that allow us to meet these requirements.

5.1 Formalizing The Problem

To solve any problem with crypto, it must first be reduced to concrete math (unless you have a ready-made crypto primitive that solves your problem for you but we don't have that). So, first, we need a way to encode our sets of groups. We do so by turning them into bit-vectors (1 for present, 0 for not present). Now the set-intersection problem becomes an inner-product problem and the consumer needs to prove that their encrypted bit-vector $\vec{b}$ has a non-zero inner product with the ACL's encrypted bit-vector $\vec{a}$. The tricky part is not revealing the bit-vectors (to either the consumer or content host) and ensuring that consumers can't use expired vectors.

To hide the vectors, DRACL uses a function-hiding inner product encryption scheme described by Bishop, Jain and Kowalczyk [16] along with the privacy improvement described by Lin (in chapter 4) [25]. This allows us to compute an inner product of two secret vectors without ever revealing anything about the vectors other than the inner product.

Note: *In this thesis, we ignore the privacy improvement ([25]) because it's a black-box modification to the underlying protocol, doesn't actually affect any of our proofs, and adds unnecessary complication and noise. The extension is literally just: double the length of the encrypted vector and fill the extra space with zeros. An implementation will have to use this but it doesn't affect any of the proofs or explanations.*

The details of this inner product encryption scheme are a bit messy so we're going to abstract them away by defining a few functions. We define the functions $\texttt{EncInput}_1$, $\texttt{EncInput}_2$, $\texttt{EncInputS}_1$, $\texttt{EncInputS}_2$, and $\texttt{EncResultS}$. $\texttt{EncInput}_i$ takes a length-two vector of cryptographic parameters and a vector to be encrypted. $\texttt{EncInputS}_i$ just takes the cryptographic parameter vector without the vector to be encrypted. $\texttt{EncResultS}$ takes a scalar.

The inner product encryption scheme can (using a function `Pair`) combine `EncInput`$_1(\vec{\alpha}, \vec{a})$ and `EncInput`$_2(\vec{\beta}, \vec{b})$ to produce `EncResultS`$(\langle\vec{\alpha}, \vec{\beta}\rangle \cdot \langle\vec{a}, \vec{b}\rangle)$ where $\vec{\alpha}$ and $\vec{\beta}$ are the length-two parameter vectors and $\vec{a}$ and $\vec{b}$ are the vectors two be encrypted. Additionally, one can combine (again, using `Pair`) `EncInputS`$_1(\vec{\alpha})$ and `EncInputS`$_2(\vec{\beta})$ to produce `EncResultS`$(\langle\vec{\alpha}, \vec{\beta}\rangle)$.

`EncInputS`, `EncInput`, and `EncResultS` all have the following properties:

1. The first argument is homomorphic over linear combinations. That is, given one of the above functions f, there is a function `LCombine` that can produce $f(x \cdot \vec{\alpha} + y \cdot \vec{\beta}, \ldots) =$ `LCombine`$(x, y, f(\vec{\alpha}, \ldots), f(\vec{\beta}, \ldots))$ assuming the remaining arguments $(\ldots)$ are identical.
2. The vectors passed into `EncInput` are "hidden". That is, given `EncInput`$_1(\ldots, \vec{a})$ and `EncInput`$_2(\ldots, \vec{b})$, it's possible to learn the inner product of these two vectors but nothing more. This was proven in [16] and [25].
3. Given `EncInput`$_1(s \cdot \vec{\alpha}, \vec{a})$, `EncInput`$_2(\vec{\beta}, \vec{b})$, `EncInputS`$_1(\vec{\alpha})$, and `EncInputS`$_2(\vec{\beta})$, it's hard to compute `EncResultS`$(s \cdot \langle\vec{\alpha}, \vec{\beta}\rangle)$ unless $\langle\vec{a}, \vec{b}\rangle$ is non-zero. We prove this in § A.4.

Now that we've abstracted away some of the underlying crypto, we can our protocol in terms of these abstractions. After that, we'll peel back the abstraction and reduce everything to math. Finally, we'll explicate define the functions used in chapter 4 in terms of the underlying crypto.

5.2 Authentication Protocol

Again, as noted in the protocol chapter (chapter 4) the protocol is abstractly simple. The content host chooses a secret scalar s, gives the consumer some encrypted form of s, and then gets back some form of s that the consumer wouldn't have been able to compute if $\langle \vec{a}, \vec{b} \rangle$ were zero. Below, we give a first attempt at the protocol and then fix some problems in the following sections. This section parallels § 4.2 but from a crypto perspective.

5.2.1 Setup

In every ACL, the producer includes $C_1 = \texttt{EncInput}_1(\vec{\alpha}, \vec{a})$ — an encryption of $\vec{a}$ — and $C_2 = \texttt{EncInputS}_1(\vec{\alpha})$.[1]

[1] (C_1, C_2) was $\texttt{Verifier}(\mathbb{A})$ in chapter 4

In the consumer's ACL Key, the producer includes $K_1 = \texttt{EncInput}_2(\vec{\beta}, \vec{b})$ [2] — an encryption of $\vec{b}$ — as the ACL Private Key and $K_2 = \texttt{EncInputS}_2(\vec{\beta})$ [3] as the ACL Public Key. Additionally, we include a certificate on K_2 certifying it as valid for some period of time.

5.2.2 Protocol

When authenticating a consumer, the content host first sends the consumer the ACL (C_1, C_2). They (the consumer) then computes:

$$\begin{aligned}
\texttt{EncResultS}(\langle\vec{\alpha}, \vec{\beta}\rangle) &= \texttt{Pair}(\texttt{EncInputS}_1(\vec{\alpha}), \texttt{EncInputS}_2(\vec{\beta})) && (1)\\
&= \texttt{Pair}(C_2, K_2)\\
\texttt{EncResultS}(c \cdot \langle\vec{\alpha}, \vec{\beta}\rangle) &= \texttt{EncResultS}(\langle\vec{\alpha}, \vec{\beta}\rangle \cdot \langle\vec{a}, \vec{b}\rangle) && (2)\\
&= \texttt{Pair}(\texttt{EncInput}_1(\vec{\alpha}, \vec{a}), \texttt{EncInput}_2(\vec{\beta}, \vec{b}))\\
&= \texttt{Pair}(C_1, K_1)
\end{aligned}$$

If $\texttt{EncResultS}(c \cdot \langle\vec{\alpha}, \vec{\beta}\rangle)$ is 1, authentication has failed and the consumer abort. This

[2] $\texttt{ProverPrivate}(\mathbb{B})$ in chapter 4
[3] $\texttt{ProverPublic}(\mathbb{B})$ in chapter 4

is the pre-verification step.

Next, the consumer sends their ACL Public Key to the content host (K_2 and the certificate on K_2) to the content host (not randomized but we'll get to that in § 5.2.3). The content host verifies the certificate and verifies that K_2 is currently valid.

The content host then computes:

$$\begin{aligned} C^s &= \texttt{EncInput}_1(s \cdot \vec{\alpha}, \vec{a}) \\ R &= \texttt{Pair}(\texttt{EncInputS}(s \cdot \vec{\alpha}),\, \texttt{EncInputS}_2(\vec{\beta})) \\ &= \texttt{EncResultS}(s \cdot \langle \vec{\alpha},\, \vec{\beta} \rangle) \end{aligned}$$

They send C^s [4] to the consumer and record R [5].

The consumer then computes:

$$\begin{aligned} \texttt{EncResultS}(sc \cdot \langle \vec{\alpha},\, \vec{\beta} \rangle) &= \texttt{EncResultS}(s \cdot \langle \vec{\alpha},\, \vec{\beta} \rangle \cdot \langle \vec{a},\, \vec{b} \rangle) \\ &= \texttt{Pair}(\texttt{EncInput}_1(s \cdot \vec{\alpha}, \vec{a}), \texttt{EncInput}_2(\vec{\beta}, \vec{b})) \\ &= \texttt{Pair}(\texttt{EncInput}_1(s \cdot \vec{\alpha}, \vec{a}), K_2) \end{aligned} \tag{3}$$

[4] $\texttt{Challenge}(s_1, \mathbb{B})$ in chapter 4

[5] $\texttt{Proof}(\texttt{Challenge}(s_1, \mathbb{B}), \texttt{ProverPublic}(\mathbb{B}))$ in chapter 4

They can then brute-force c because it must be small (it's the size of the intersection) by repeatedly using `LCombine` on $\texttt{EncResultS}(\langle\vec{\alpha},\, \vec{\beta}\rangle)$ (which they learned during pre-verification). That is, they repeatedly call the following until they find a $\overset{?}{c}$ that works:

$$\texttt{LCombine}(\overset{?}{c}, 0, \texttt{EncResultS}(\langle\vec{\alpha},\, \vec{\beta}\rangle), \texttt{EncResultS}(\langle\vec{\alpha},\, \vec{\beta}\rangle)) \overset{?}{=} \texttt{EncResultS}(c \cdot \langle\vec{\alpha},\, \vec{\beta}\rangle)$$

They then use this c and (3) to compute $\texttt{EncResultS}(s \cdot \langle\vec{\alpha},\, \vec{\beta}\rangle)$ [6] and prove to the content host that they know this value. Given propriety 3 from § 5.1, this would have been hard if the two vectors had a zero inner product.

Unfortunately, the protocol as describe so far doesn't meet our anonymity requirement. $\vec{\beta}$ and, in-turn, K_2 uniquely identify the consumer.

5.2.3 Anonymity

To hide the consumer's identity, we randomize[7] the consumer's ACL Public Key in such a way that still allows the consumer to take the inner product between $\vec{a}$ and $\vec{b}$. To do so, we give the consumer two keys that encode the same vector but

[6] $\texttt{Proof}(\texttt{Challenge}(s_1, \mathbb{B}), \texttt{ProverPublic}(\mathbb{B}))$ in chapter 4
[7] `Randomize` in chapter 4

have different $\vec{\beta}$'s and allow them to take linear combinations. Linear combinations randomize $\vec{\beta}$ and hide the user's identity. That is, given:

$$K_1 = \mathtt{EncInput}_2(\vec{\beta}, \vec{b}) \qquad K_2 = \mathtt{EncInputS}_2(\vec{\beta})$$

$$K_1' = \mathtt{EncInput}_2(\vec{\beta}', \vec{b}) \qquad K_2' = \mathtt{EncInputS}_2(\vec{\beta}')$$

Compute (for some random x and y):

$$K_1^r = \mathtt{EncInput}_2(\langle x\vec{\beta},\ y\vec{\beta}'\rangle, \vec{b}) \qquad = \mathtt{LCombine}(x, y, K_1, K_1')$$

$$K_2^r = \mathtt{EncInput}_2(\langle x\vec{\beta},\ y\vec{\beta}'\rangle) \qquad = \mathtt{LCombine}(x, y, K_2, K_2')$$

And use K_1^r and K_2^r instead of K_1 and K_2.

5.2.4 Expiration/Authenticity

Unfortunately, now that we're allowing the user to take linear combinations of their keys, we can't just sign K_2 (because the consumer never gives K_2 to the content host). Instead we need a malleable signature scheme that allows consumers to take linear combinations of (the parameters of) K_2 while preserving the signature.

Luckily, Signatures For Network Coding[20] proposes a scheme that almost gives us what we want. The specific scheme proposed assumes that the vectors will be in the clear, not encrypted like in our scheme. However, some modifications make this scheme work with DRACL. We don't discuss them here because the details rely on the specific definitions of the abstract functions we've been using. Instead, you can find the details in § 5.4.

5.3 Tearing Down The Abstraction

Finally, we can tear down our abstraction. Luckily, it maps directly to the underlying function-hiding inner product encryption crypto.

First, whenever we have vectors in exponents of groups, we mean a vector of groups to the elements of the vector. That is, $g_i^{\vec{v}} = (g_i^{v_1}, g_i^{v_2}, \ldots, g_i^{v_n})$.

Then, given the vectors $\vec{d}$, $\vec{d^\star}$, $\vec{b}$, and $\vec{b^*}$ as described in [16] and a bilinear pairing function $\mathbf{e}$, the functions we've been using map to the underlying crypto protocol as described in § 5.3. Additionally, unlike [16], we scale the $\vec{b^*}_{2n+1}$ and $\vec{b^*}_{2n+3}$ in $\mathtt{EncInput}_1(\vec{\alpha}, \vec{x})$ by random independent r_1 and r_2 factors chosen fresh for each encryption of $\vec{x}$. This does not affect privacy (proof in § A.5) but allows our security proof to go through.

$$\begin{aligned}
\texttt{EncInputS}_1(\vec{\alpha}) &= g_1^{\alpha_1 \vec{d^*}_1 + \alpha_2 \vec{d^*}_2} \quad \text{// In [16], } \alpha_1 = \alpha \text{ and } \alpha_2 = \tilde{\alpha}. \\
\texttt{EncInputS}_2(\vec{\beta}) &= g_2^{\beta_1 \vec{d}_1 + \beta_2 \vec{d}_2} \quad \text{// In [16], } \beta_1 = \beta \text{ and } \beta_2 = \tilde{\beta}. \\
\texttt{EncInput}_1(\vec{\alpha}, \vec{x}) &= g_1^{\alpha_1 \left(\sum_{x_i \in \vec{x}} x_i \vec{b^*}_i + r_1 \vec{b^*}_{2n+1}\right) + \alpha_2 \left(\sum_{x_i \in \vec{x}} x_i \vec{b^*}_{n+i} + r_2 \vec{b^*}_{2n+3}\right)} \\
\texttt{EncInput}_2(\vec{\beta}, \vec{x}) &= g_2^{\beta_1 \left(\sum_{x_i \in \vec{x}} x_i \vec{b}_i + \vec{b}_{2n+2}\right) + \beta_2 \left(\sum_{x_i \in \vec{x}} x_i \vec{b}_{n+i} + \vec{b}_{2n+4}\right)} \\
\texttt{Pair}(a, b) &= \mathbf{e}(a, b) \\
\texttt{EncResultS}(a) &= \mathbf{e}(g_1, g_2)^a \\
\texttt{LCombine}(x, y, g_i^{\vec{u}}, g_i^{\vec{v}}) &= \left(g_i^{\vec{u}}\right)^x \cdot \left(g_i^{\vec{v}}\right)^y = g_i^{x\vec{u} + y\vec{v}}
\end{aligned}$$

Figure 5.1: Cryptographic Abstraction

5.4 Malleable Signature Scheme

Now that we've mapped our crypto onto well-known cryptographic primitives (i.e. cryptographic pairing), we can explain the malleable signature scheme. Specifically, we need to sign a set of $K_2 = \texttt{EncInputS}_2(\vec{\beta}) = g_2^{\beta_1 \vec{d}_1 + \beta_2 \vec{d}_2}$ such that the signature is preserved under linear combination.

To do this, we note that K_2 is of the form $(g_2^{x_1}, g_2^{x_2})$ for some x_1 and x_2. Given two K_2's, we want the signature to be preserved under linear combination. That is, $(g_2^{ax_1+bx'_1}, g_2^{ax_2+bx'_2})$ for some a and b.

For our signature scheme, we have the producer pick some o_1 and o_2, release $P = (g_1^{o_1}, g_1^{o_2})$ as a public key. The signature on K_2 is then $\sigma = (g_2^{x_1 \cdot o_1}, g_2^{x_2 \cdot o_2})$. To verify the signature, the content host verifies that $\mathbf{e}(g_1, \sigma_i) = \mathbf{e}(g_1^{o_i}, g_2^{x_i})\ \forall i$. This is loosely based on [20]. Note, we keep the signatures on the individual elements separate (unlike in [20]) because we have slightly different requirements (and we needed to do so to make the proof go through). In future work, we hope to follow [20] more closely.

Finally, when creating the certificate validating the ACL Key for a time period, the producer/AP certify P, not K_2 directly.

5.5 Putting It Together

Finally, we can tie these two chapters together by explicitly defining the functions used in chapter 4 with those defined here. This section is mostly for implementors.

5.5.1 Signature Scheme

Private Key `HSigPrivate` $= (o_1, o_2)$ (randomly generate)

Public Key `HSigPublic` $= (g_1^{o_1}, g_1^{o_2})$

Signing

Given $(g_2^{x_1}, g_2^{x_2})$, produce a signature $(\sigma_1, \sigma_2) = (g_2^{x_1 \cdot o_1}, g_2^{x_2 \cdot o_1})$.

Verification

Given $(g_2^{x_1}, g_2^{x_2})$ and (σ_1, σ_2), verify that $\mathbf{e}(g_1^{o_i}, g_2^{x_i}) = \mathbf{e}(g_1, \sigma_i)\ \forall i \in (1, 2)$.

5.5.2 Encrypted Vectors

Given some bit-vector $\vec{x}$ describing the groups that have access to a resource and randomly chosen cryptographic parameters $\vec{\alpha}$, we define $\texttt{Verifier}(\mathbb{A})$ as follows:

$$\texttt{Verifier}(\mathbb{A}) = (\texttt{EncInputS}_1(\vec{\alpha}), \texttt{EncInput}_1(\vec{\alpha}, \vec{x}))$$

Given some bit-vector $\vec{y}$ describing the groups that a consumer is in and randomly chosen cryptographic parameters $\vec{\beta}$, we define $\texttt{ProverPublic}(\mathbb{B})$ and $\texttt{ProverPrivate}(\mathbb{B})$ as follows:

$$\texttt{ProverPublic}(\mathbb{B}) = \texttt{EncInputS}_2(\vec{\beta})$$

$$\texttt{ProverPrivate}(\mathbb{B}) = \texttt{EncInput}_2(\vec{\beta}, \vec{y})$$

5.5.3 Key Generator

As described in § 5.2.3, we allow the consumer to generate unlinkable key-pairs as follows.

First, we define $\texttt{ProverGenerator}(\mathbb{B})$ to be the tuple (K_1, K_2, K_1', K_2') (again with random $\vec{\beta}$ parameters):

$$\begin{aligned} K_1 &= \texttt{EncInput}_2(\vec{\beta}, \vec{b}) & K_2 &= \texttt{EncInputS}_2(\vec{\beta}) \\ K_1' &= \texttt{EncInput}_2(\vec{\beta}', \vec{b}) & K_2' &= \texttt{EncInputS}_2(\vec{\beta}') \end{aligned}$$

We then define $\texttt{Randomize}(\vec{r}, G, \texttt{HSigGenerator}(\texttt{HSigPublic}, G), \ldots)$ to output the tuple $(K_1^r, K_2^r, \sigma_i, \ldots)$ define as:

$$\begin{aligned} K_1^r &= \texttt{ProverPublic}(\mathbb{B}) && = \texttt{LCombine}(r_1, r_2, K_1, K_1') \\ K_2^r &= \texttt{ProverPrivate}(\mathbb{B}) && = \texttt{LCombine}(r_1, r_2, K_2, K_2') \\ \sigma_i &= \texttt{HSig}(\texttt{HSigPublic}_i, K_2^r) && = (K_2^r)^{\texttt{HSigPrivate}_i} \end{aligned}$$

5.5.4 Pre-Verification

We define the following method for pre-verifying access. This will spit out the size of the intersection c if we have access. c will be re-used in `MakeProof` so hang onto it. We didn't explicitly mention this in chapter 4 because it's technically an optimization.

Compute:

$$\begin{aligned}
\texttt{EncResultS}(\langle\vec{\alpha},\,\vec{\beta}\rangle) &= \texttt{Pair}(\texttt{EncInputS}_1(\vec{\alpha}), \texttt{EncInputS}_2(\vec{\beta})) && (1)\\
&= \texttt{Pair}(C_2, K_2)\\
\texttt{EncResultS}(c\cdot\langle\vec{\alpha},\,\vec{\beta}\rangle) &= \texttt{EncResultS}(\langle\vec{\alpha},\,\vec{\beta}\rangle\cdot\langle\vec{a},\,\vec{b}\rangle) && (2)\\
&= \texttt{Pair}(\texttt{EncInput}_1(\vec{\alpha},\vec{a}), \texttt{EncInput}_2(\vec{\beta},\vec{b}))\\
&= \texttt{Pair}(C_1, K_1)
\end{aligned}$$

And then brute-force c using `LCombine` as described in § 5.2.2.

$\texttt{PreVerify}(\texttt{Verifier}(\mathbb{A}), \texttt{ProverPublic}(\mathbb{B}), \texttt{ProverPrivate}(\mathbb{B})) = c \quad 0 \rightarrow$ no access

5.5.5 Challenge Generation

$$\begin{aligned}
\texttt{MakeChallenge}(s_1, \texttt{ProverPublic}(\mathbb{B})) &= \texttt{Verifier}(\mathbb{A})_1^s \\
&= \texttt{EncInput}_1(s \cdot \vec{\alpha}, \vec{a}) \\
\texttt{PredictProof}(\texttt{Verifier}(\mathbb{A}), s, \texttt{ProverPublic}(\mathbb{B})) &= \texttt{Pair}(\texttt{Verifier}(\mathbb{A})_2^s,\ \texttt{ProverPublic}(\mathbb{B})) \\
&= \texttt{Pair}(\texttt{EncInputS}(s \cdot \vec{\alpha}),\ \texttt{EncInputS}_2(\vec{\beta})) \\
&= \texttt{EncResultS}(s \cdot \langle \vec{\alpha},\ \vec{\beta} \rangle)
\end{aligned}$$

5.5.6 Proof Generation

Where c is computed by `PreVerify` (§ 5.5.4).

$$\begin{aligned}
&\texttt{MakeProof}(\\
&\qquad\qquad \texttt{Challenge}(s_1, \mathbb{B}), \\
&\qquad\qquad c, \\
&\qquad\qquad \texttt{ProverPrivate}(\mathbb{B}) \\
&\qquad) = \texttt{Pair}(\texttt{Challenge}(s_1, \mathbb{B}), \texttt{ProverPrivate}(\mathbb{B}))^{-c}
\end{aligned}$$

Chapter 6

API

This chapter covers how to use DRACL as a content-host developer.

6.1 Server Side API

We expose two functions as a part of the server-side API.

λ `is_acl(acl) -> bool`

Validates that the ACL is well-formed. The content host should call this method before storing an ACL to catch errors up-front.

λ `check_access(server_secret, origin, acl, response) -> Challenge`

Runs a round in the authentication protocol.

The parameters are:

`server_secret`

> A server-specific symmetric secret key. This needs to stay the same for the course of the authentication protocol (multiple rounds of communication) however, it can be re-used everywhere. Your web framework likely already *has* a global server secret. Just use that one.

`origin`

> The content host's origin. This allows the content host to verify that the no-one is performing a man-in-the-middle attack on the consumer.

`acl`

> The ACL.

`response`

> An opaque `response` from the consumer

It returns one of:

`Grant(until)`

> Grant the consumer access until `until`. The `until` field dictates until

when the content host should grant access. This allows content hosts to to record, in a cookie for example, how long the user should be granted access to the resource without having to re-authenticate.

`Continue(challenge)`

Return `challenge` to the consumer.

`Deny`

Deny access.

6.1.1 Client Side API

The client-side API uses JavaScript. We'd liked to have supported a variant that uses HTTP-Auth headers but, unfortunately, the challenge object used in DRACL is large ($\approx$ 128 KiB) and simply wouldn't fit in an HTTP header. Fortunately, the response object is tiny (end-users tend to have little upstream bandwidth).

The client exposes several functions:

λ `authenticate(challenge, callback: function(response))`

This function takes an opaque challenge blob (or the ACL if this is the first round of authentication). It calls callback if the consumer decides to proceed.

Note: ***`callback`*** *will never be called if the user chooses to not authenticate. See section § 3.1.2.2.*

The `response` argument to `callback` should be sent back to the server and run through the server-side `check_access` function.

λ `create_acl(description, callback: function(acl))`

Ask the browser to create an ACL. The `description` parameter is a human readable description of the purpose of this ACL.

λ `edit_acl(acl, callback: function(acl))` Asks the browser to modify an ACL. This is how a producer edits who can access a piece of content.

6.1.2 Authentication Protocol

From the developer's perspective, the protocol works as follows:

1. The consumer attempts to access a piece of content protected by an ACL.
2. `server` The content host returns the ACL to the consumer's browser as `challenge`
3. `client` The content host calls `authenticate` in the consumer's browser:

```
authenticate(challenge, function(response) {
  /* ... */
});
```

4. <u>`client`</u> When and if the callback executes, it should return `response` to the content host.
5. <u>`server`</u> On the content host, call `check_access(secret, origin, acl, resp)`.
6. <u>`server`</u> If the result is `Deny`, abort.
7. <u>`server`</u> If the result is `Continue(challenge)`, send `challenge` back to the consumer's browser and go back to step 3.
8. <u>`server`</u> If the result is `Grant(until)`, grant access until the time specified by `until`.

Importantly, the server maintains no mutable state.

Bibliography

[1] Facebook connect user friends permission. https://developers.facebook.com/docs/facebook-login/permissions#reference-user_friends.

[2] Mozilla persona. https://login.persona.org/.

[3] Nacl crypto library.

[4] Openid authentication 2.0 - final. http://openid.net/specs/openid-authentication-2_0.html.

[5] Plaintext offenders. http://plaintextoffenders.com/.

[6] pump.io. http://pump.io/.

[7] Tails: The amnesic incognito live system. https://tails.boum.org/.

[8] Tor project. https://torproject.org/.

[9] User managed access (uma). http://kit.mit.edu/projects/user-managed-access-uma.

[10] Webid. https://www.w3.org/2005/Incubator/webid/spec.

[11] Ftc charges deceptive privacy practices in googles rollout of its buzz social network.

[12] Identity/persona aar, Mar 2014.

[13] Aas, J. Progress towards 100% https, june 2016.

[14] Beuchat, J.-L., González-Díaz, J. E., Mitsunari, S., Okamoto, E., Rodríguez-Henríquez, F., and Teruya, T. High-speed software implementation of the optimal ate pairing over barreto–naehrig curves. In *International Conference on Pairing-Based Cryptography* (2010), Springer, pp. 21–39.

[15] Birgisson, A., Politz, J. G., lfar Erlingsson, Taly, A., Vrable, M., and Lentczner, M. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Network and Distributed System Security Symposium* (2014).

[16] Bishop, A., Jain, A., and Kowalczyk, L. Function-hiding inner product encryption. In *International Conference on the Theory and Application of Cryptology and Information Security* (2015), Springer, pp. 470–491.

[17] Bode, K. Opting out of at&t's 'gigapower' snooping is comically expensive.

[18] Bode, K. Comcast says it wants to charge broadband users more for privacy.

[19] Carminati, B., Ferrari, E., and Perego, A. Rule-based access control for social networks. In *On the Move to Meaningful Internet Systems 2006: OTM 2006 Workshops*, R. Meersman, Z. Tari, and P. Herrero, Eds., vol. 4278 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006, pp. 1734–1744.

[20] Charles, D., Jain, K., and Lauter, K. Signatures for network coding. *International Journal of Information and Coding Theory 1*, 1 (2009), 3–14.

[21] Florencio, D., and Herley, C. A large-scale study of web password habits. In *Proceedings of the 16th International Conference on World Wide Web* (New York, NY, USA, 2006), WWW '07, ACM, pp. 657–666.

[22] Freudenthal, E., Pesin, T., Port, L., Keenan, E., and Karamcheti, V. drbac: distributed role-based access control for dynamic coalition environments. In *Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on* (2002), pp. 411–420.

[23] Hammer-Lahav, E. E. The OAuth 1.0 Protocol. RFC 5849, April 2010.

[24] K. Zeilenga, E. Lightweight directory access protocol (ldap): Technical specification road map. RFC 4510.

[25] Lin, H. Indistinguishability obfuscation from constant-degree graded encoding schemes. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques* (2016), Springer, pp. 28–57.

[26] Neuman, B., and Ts'o, T. Kerberos: an authentication service for computer networks. *Communications Magazine, IEEE 32*, 9 (Sept 1994), 33–38.

[27] RSnake. Slowloris http dos. `https://web.archive.org/web/20090822001255/http://ha.ckers.org:80/slowloris/`, June 2009.

[28] Ruj, S., Stojmenovic, M., and Nayak, A. Decentralized access control with anonymous authentication of data stored in clouds. *IEEE Transactions on Parallel and Distributed Systems 25*, 2 (2014), 384–394.

[29] US-CERT. Dns amplification attacks. Alert TA13-088A, Mar. 2013.

[30] Van Den Hooff, J., Lazar, D., Zaharia, M., and Zeldovich, N. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles* (2015), ACM, pp. 137–152.

Appendices

Appendix A

Proofs

Gandalf: Fly, you fools!

A.1 Consumer Anonymity

In this section, we prove our anonymity requirement. Specifically, we prove that the content learns nothing more than whether or not the consumer has access to a piece of content. We assume that neither the producer nor the AP collude with the content host. Otherwise, this would be theoretically unprovable for any access control system as the producer could create an ACL that grants access to precisely one consumer.

First, the consumer re-randomizes their ACL Public Key before authenticating so the content host cannot identify the consumer though this public key. As a matter of fact, the re-randomization mechanism we use information theoretically hides the consumer's identity. The consumer re-randomizes their ACL Public Key by taking linear combinations of two length-two linearly independent vectors. As these vectors are linearly independent, they form a basis and a random linear combination of any two of these vectors is truly random.

Second, it's impossible for the content host to learn anything other than whether or not the given consumer is listed in the ACL because they can't cheat in the authentication protocol. Basically, they can ask a single question ("are you (the consumer) a member of this ACL?") signed by the producer and learns the hash of the secret that was used to generate the challenge. Proof: recall that after successfully generating a challenge-response, the consumer learns the secret that the content host used when generating the challenge. Also recall that the consumer learns the ACL (signed by the producer). This means that the consumer can re-create the challenge and verify that it was created correctly (well, that it could have been created correctly).

This tells the content host precisely that the consumer was able to extract the secret from the challenge (the bit that indicates that the user should be granted access) and nothing more (they already must have known the secret to produce the challenge).

Therefore, at the end of the authentication protocol, the content host learns only that the consumer is a member of the ACL.

□

A.2 Producer Privacy

In this section, we prove privacy. That is, we prove that DRACL hides the structure of its producers' social networks from third parties. Specifically, we prove that no party can directly learn which groups are listed in any given ACL or which groups a consumer is a member of (although consumers can learn the size of the intersection between their groups and those listed in an ACL).

Proof: This follows directly from the guarantee given by the underlying function-hiding inner product encryption scheme [16]: the function-hiding inner product encryption scheme reveals nothing about the "functions" (vectors) except their inner products (the size of the intersection).

$\square$

A.3 Malleable Signature Scheme

In DRACL, we use a malleable signature scheme that has the following property: Given a set of uniformly random vectors (unknown to the attacker) in exponents and a set of signatures on these vectors an attacker cannot forge a signature for a known vector in exponents. This is a very weak guarantee but it turns out it's all we need.

Recall our signature scheme:

Private Key $(o_i, \ldots)$ (random)

Public Key $(g_1^{o_i}, \ldots)$

Signing

Given $(g_2^{x_i}, \ldots)$, produce a signature $(\sigma_i, \ldots) = (g_2^{x_i \cdot o_i}, \ldots)$.

Verification

Given $(g_2^{x_i}, \ldots)$ and $(\sigma_1, \ldots)$, verify that $\mathbf{e}(g_1^{o_i}, g_2^{x_i}) = \mathbf{e}(g_1, \sigma_i)\ \forall i$.

We prove this element-wise.

A.3.1 Assumption

It's hard to compute g_1^b or g_2^b given:

$$g_1^a g_1^{ab}$$

$$g_2^a g_2^{ab}$$

This is a variant of SXDH.

A.3.2 Theorem

Assume there exists a function f that can take a set of valid value/signature pairs with unknown random values and spit out a signature and a known value.

Let, for some x, o, and p:

$$g_1^o = g_1^a g_2^x = g_2^b \qquad\qquad g_2^{xo} = g_2^{ab}$$

o is the private key, (g_1^o, g_1) is the public key.

We can generate as many signed elements as we want by raising g_2^x and g_2^{xo} to some power of our choice r. We pass all these elements to f which spits out g_2^v and g_2^{vo}. We can now compute g_2^o which violates our assumption. Therefore, f cannot exist.

A.4 Security

In this section, we prove that DRACL is secure. That is, no party can convince a content host that they are listed in an ACL unless they have an ACL Private Key whose group set intersects with the ACL's group set and an associated ACL Public key with a valid certificate.

A.4.1 Assumption

To prove security, we assume that to compute $(\mathbf{e}(g_1, g_2)^{abc}, g_2^a)$ for some non-degenerate a given:

$$g_1, g_1^b, g_1^{c^{-1}}$$

$$g_2, g_2^c$$

One must know a.

Another way of putting this is that, if there exists a function that can compute $(\mathbf{e}(g_1, g_2)^{abc}, g_2^a)$ given the variables above for some non-degenerate a but not tell us what a is, then there must exist an extractor function that can compute this *and* tell us what a is.

A.4.2 Theorem

Given all information known by parties other than the content owner *except* the set of private keys whose group vectors intersect some set of target ACLs chosen by the attacker, it is hard for an attacker to prove against any of the target ACLs.

Specifically, the attacker can:

- Pick which groups can access which resources ($\mathbf{X}$) and which users are in which groups ($\mathbf{Y}$), both described by binary vectors.
- Pick some set of target ACLs against which they intend to prove access.
- Know all ACLs.
- Know all user public keys and all public key generators for all epochs.
- Know all user secret keys for all epochs *except* those valid in the current epoch.
- Know all user secret keys valid in current epoch *except* those that grant access to the target ACL.

A.4.2.1 Reduction

Assume there exists a function f that can take all of the information listed above and return a proof of access against the target ACL.

There exists an efficient reduction from the security assumption to f:

First, generate some random element in g_2 and call it g_2^t (importantly we don't know t). Then, let:

$$g_1^{sq} = g_1^b \qquad g_2^{q^{-1}} = g_2^c \qquad g_1^q = g_1^{c^{-1}}$$

Then:

1. Pick $\mathbf{X} = \{\vec{x}_i\}$ (where each vector describes the groups that have access to a resource) and $\mathbf{Y} = \{\vec{y}_i\}$ (where each vector describes the groups in which a consumer is a member) such that there exists some inner product $\langle \vec{y}_i, \vec{x}_j \rangle = 0$ (some consumer does not have access to some resource).
2. Pick some non-empty set of vectors $\mathbf{X}_\tau$ of $\mathbf{X}$ to represent the ACLs of the

target resource(s).

3. Let γ be the set of groups the attacker is a member of in the current epoch. Choose a subset $\mathbf{Y}_\upsilon$ of the vectors in $\mathbf{Y}$ that have a zero inner products with the rows in $\mathbf{X}_\tau$. *not* a member. $\gamma = \{j \mid \exists\, \vec{y} \in \mathbf{Y}_\upsilon.\ \vec{y}_i[j] = 1\}$ (i.e., the attacker is a member in the j^{th} group in one of their ACL keys).
4. Pick $\mathbb{D}, \mathbb{D}^* \leftarrow \mathbf{Dual}(\mathbb{Z}_p^2)$, and $\mathbb{B}', \mathbb{B}'^* \leftarrow \mathbf{Dual}(\mathbb{Z}_p^{2n+4})$ as described in the paper. $(\mathbb{B}', \mathbb{B}'*)$ are the same as $(\mathbb{B}, \mathbb{B}'*)$ as described in the paper except that the vectors that match the groups that the attacker is not in are scaled by a factor of q^{-1}.
5. Generate all ACLs in the system: For each vector in $\mathbf{X}$,
   (a) Pick α, $\tilde{\alpha}$ as described in the paper.
   (b) Pick r_1 and r_2 randomly.
   (c) Let

$$\begin{aligned} C_2 &= g_1^{\alpha \vec{d_1^*} + \tilde{\alpha} \vec{d_2^*}} \\ C_1 &= (g_1^q)^{\alpha\left(\sum_{\forall x_i \in \vec{x} | i \in \bar{\gamma}} x_i \vec{b^*}_i + r_1 \vec{b^*}_{2n+1}\right) + \tilde{\alpha}\left(\sum_{\forall x_i \in \vec{x} | i \in \bar{\gamma}} x_i \vec{b^*}_i + r_2 \vec{b^*}_{2n+3}\right)} \\ &\quad \times (g_1)^{\alpha\left(\sum_{\forall x_i \in \vec{x} | i \in \gamma} x_i \vec{b^*}_i\right) + \tilde{\alpha}\left(\sum_{\forall x_i \in \vec{x} | i \in \gamma} x_i \vec{b^*}_{n+i}\right)} \end{aligned}$$

   (d) If this is a target ACL, generate the target challenge:

$$C_s = (g_1^{sq})^{\alpha\left(\sum_{x_i \in \vec{x}} x_i \vec{b^*}_i + r_1 \vec{b^*}_{2n+1}\right) + \tilde{\alpha}\left(\sum_{x_i \in \vec{x}} x_i \vec{b^*}_i + r_2 \vec{b^*}_{2n+3}\right)}$$

(e) Otherwise, generate a custom challenge with known s'. Let

$$C_{s'} = C_1^{s'}$$

6. Generate keys for all epochs other than the current one.

   (a) First generate the keys:

$$\begin{aligned} K_2 &= g_2^{\beta\vec{d_1}+\tilde{\beta}\vec{d_2}} \\ K_1 &= \left(g_2^{-q}\right)^{\alpha\left(\sum_{\forall y_i \in \vec{y} | i \in \bar{\gamma}} y_i \vec{b}_i\right)+\tilde{\alpha}\left(\sum_{\forall y_i \in \vec{y} | i \in \bar{\gamma}} y_i \vec{b}_i\right)} \\ &\quad \times \left(g_2\right)^{\alpha\left(\sum_{\forall y_i \in \vec{y} | i \in \gamma} y_i \vec{b}_i + \vec{b}_{2n+1}\right)+\tilde{\alpha}\left(\sum_{\forall y_i \in \vec{y} | i \in \gamma} y_i \vec{b^*}_{n+i} + \vec{b}_{2n+3}\right)} \end{aligned}$$

   (b) Then pick a random o_1 and o_2 as the malleable signature scheme secret key for the old epoch(s) and then generate the public key: $P = g_1^{(o_1, o_2)}$.

   (c) Finally, malleable sign the K_2's for the old epoch(s) with the old malleable signature keys:

$$\begin{aligned} \sigma(K_2) &= (K_{2_1}^{o_1}, K_{2_2}^{o_2}) \\ \sigma(K_2') &= ({K_{2_1}'}^{o_1}, {K_{2_2}'}^{o_2}) \end{aligned}$$

7. Generate keys for the current epoch for all consumers in $\mathbf{Y}_\upsilon$:

(a) First, generate the keys where $\beta = t \cdot \beta^{\dagger}$ and $\tilde{\beta} = t\tilde{\beta^{\dagger}}$:

$$K_2^{\tau} = \left(g_2^t\right)^{\beta^{\dagger}\vec{d_1}+\tilde{\beta^{\dagger}}\vec{d_2}}$$

$$K_1^{\tau} = \left(g_2^t\right)^{\beta^{\dagger}\left(\sum_{y_i \in \vec{y}} y_i\vec{b_i}+\vec{b}_{2n+1}\right)+\tilde{\beta^{\dagger}}\left(\sum_{y_i \in \vec{y}} y_i\vec{b_i}+\vec{b}_{2n+3}\right)}$$

$$K_2^{\tau}{}' = \left(g_2^t\right)^{\beta^{\dagger}\vec{d_1}+\tilde{\beta^{\dagger}}\vec{d_2}}$$

$$K_1^{\tau}{}' = \left(g_2^t\right)^{\beta^{\dagger}\left(\sum_{y_i \in \vec{y}} y_i\vec{b_i}+\vec{b}_{2n+1}\right)+\tilde{\beta^{\dagger}}\left(\sum_{y_i \in \vec{y}} y_i\vec{b_i}+\vec{b}_{2n+3}\right)}$$

(b) Then pick a random o_1^{τ} and o_2^{τ} as the malleable signature secret key for the current epoch and then generate the public key: $P^{\tau} = g_1^{(o_1^{\tau}, o_2^{\tau})}$.

(c) Finally, sign the K_2^{τ}'s with the malleable signature keys for the current epoch:

$$\sigma(K_2^{\tau}) = ({K^{\tau}}_{2_1}^{o_1}, {K^{\tau}}_{2_2}^{o_2})$$

$$\sigma(K_2^{\tau}{}') = ({K^{\tau}{}'}_{2_1}^{\,o_1}, {K^{\tau}{}'}_{2_2}^{\,o_2})$$

We feed all K_1's and K_2's, all C_1's and C_2's, all challenges (C_s, $C_{s'}$), the malleable signature public keys (P), and the signatures on K_2's from the all epochs ($\sigma(K_2)$ into f.

To break our scheme, f would have to produce (where $\vec{z}$ is some length two vector):

$$K_2^{(target)} = (g_2)^{\vec{z}}$$

$$R = \mathbf{e}(C_1, K_2^{(target)})^s$$

Additionally, it would have to signature $\sigma_k = \sigma(K_2^{(target)})$ such that $\mathbf{e}(K_2^{(target)}, P^\tau) == \sigma_k$ (i.e., the signature verifies).

WLOG, we can factor $K_2^{(target)}$ and R into the form:

$$K_2^{(target)} = g_2^{z_1 \vec{d_1^*} + z_2 \vec{d_2^*}} \qquad \vec{d_1^*} \text{ and } \vec{d_2^*} \text{ form a basis}$$

$$R = \mathbf{e}(g_1, g_2)^{s \cdot (\alpha z_1 + \tilde{\alpha} z_2)}$$

Given that we know α, $\tilde{\alpha}$, $\vec{d_1^*}$, and $\vec{d_2^*}$ (we chose them), we can solve for $g_2^{\alpha z_1 + \tilde{\alpha} z_2}$ (without needing to know the z's). With $a = \alpha z_1 + \tilde{\alpha} z_2$, we now have $\mathbf{e}(g_1, g_2)^{abc}$ (recall $b = sq$ and $c = q^{-1}$) and g_2^a for some a.

However, we can't possibly know the value of a. Recall that, given the malleable signature scheme we're using, it's computationally infeasible to compute a signature on a chosen vector. Given that we don't know the exponents of the vectors we signed with P^τ, we cannot produce a vector signed with P^τ with known exponents. Therefore, regardless of the vector that f returns, if P^τ verifies, we (the reduction and f) are guaranteed to not know the exponents. Therefore, we're guaranteed to not know (z_1, z_2) which, in turn, means we can't know a.

However, this violates our assumption so f cannot exist.

$\square$

A.5 Inner-Product Encryption Modification Proof

Our modification to the inner-product encryption ([16]) construction does not affect its security/privacy guarantees.

Recall that, unlike [16], we scale the $\vec{b^*}_{2n+1}$ and $\vec{b^*}_{2n+3}$ vectors in $\texttt{EncInput}_1(\vec{\alpha}, \vec{x})$ by random independent r_1 and r_2 factors chosen fresh for each encryption of $\vec{x}$.

This does not affect correctness because these vectors drop out in the inner product.

This does not affect the security of the inner-product encryption scheme because an attacker could do the same thing after we construct the ciphertexts and keys (C_1, C_2, K_1, K_2). Specifically, given two C_1's (v_0, v'_0) that encode zero vectors (constructed as described in [16]) and an arbitrary number of C_1's (v_t) where we want to randomize $\vec{b^*}_{2n+1}$ and $\vec{b^*}_{2n+3}$, we can:

1. Take a linear combination of l v_0's and m v'_0's to get (all the other b^*'s drop out because our vector is zero):

$$v_0^{(r)} = g_1^{(l\alpha_0 + m\alpha'_0)\vec{b^*}_{2n+1} + (l\tilde{\alpha}_0 + m\tilde{\alpha}'_0)\vec{b^*}_{2n+3}}$$

2. "add" this $v_0^{(r)}$ to the target vector (v_t) and observe that r_1 and r_2 are completely random.

$$\begin{aligned} v_t^{(r)} &= g_1^{\alpha_t(\sum_{x_i \in \vec{x}} x_i \vec{b^*}_i + (\alpha_t)^{-1}(l\alpha_0 + m\alpha'_0)\vec{b^*}_{2n+1}) + \tilde{\alpha}_t(\sum_{x_i \in \vec{x}} x_i \vec{b^*}_{n+i} + (\tilde{\alpha}_t)^{-1}(l\tilde{\alpha}_0 + m\tilde{\alpha}'_0)\vec{b^*}_{2n+3})} \\ &= g_1^{\alpha_t(\sum_{x_i \in \vec{x}} x_i \vec{b^*}_i + r_1 \vec{b^*}_{2n+1}) + \tilde{\alpha}_t(\sum_{x_i \in \vec{x}} x_i \vec{b^*}_{n+i} + r_2 \vec{b^*}_{2n+3})} \\ r_1 &= (\alpha_t)^{-1}(l\alpha_0 + m\alpha'_0) \\ r_2 &= (\tilde{\alpha}_t)^{-1}(l\tilde{\alpha}_0 + m\tilde{\alpha}'_0) \end{aligned}$$

As we were able to add these random factors after the fact, it can't possibly affect privacy (except that an attacker learns two encryptions of the zero vector).

□

A.6 The Producer Is Public Proofs

This section is simply a set of expanded proofs for the claims we made in § 3.1.1.2.

A.6.1 Option 4 Is Provably Impossible

Here, we prove that option 4 from § 3.1.1.2 is impossible to achieve in any protocol that meets DRACL's performance requirements. Forgetting DRACL, assume there exists a system that meets the requirements of DRACL. This system must avoid updating individual ACLs when the members of a group change (requirement 1). This system must avoid contacting a third party such as the AP while authenticating (requirement 2). To achieve the first requirement, the ACLs must not fully specify who has access to what (there must be some indirection). Therefore, there must be some additional information needed for authentication that describes the current state of the producer's social network (i.e., what consumers are in what groups). Additionally, this information must be specific to the producer (it describes the producer's social network). To achieve the second requirement, this additional information must be reusable between authentication attempts against multiple ACLs because only the consumer and the content host are allowed to participate in the authentication protocol. Therefore, given an ACL known to have been authored by some producer and another ACL authored by an unknown producer, either the consumer or the content host (the only participating parties) must learn whether or not the second ACL was authored by the same producer as the first simply because it will reuse the same producer specific information.

□

A.6.2 Option 2 Requires A Linear Search

We asserted that the producer would have to linearly search through a list of producer-specific secrets to provide guarantee 2, here we give a proof sketch (with an assumption) of why we believe this to be true. Basically, the ACL would have to include some constant-sized tag that only an authorized consumer could recognize. To be constant-sized, it can't describe the list of consumers that should be able to recognize it (would require infinite compression). Therefore, this information has to be distributed among the consumers (it has to exist somewhere). Therefore, every consumer would have some tag-recognition key from that corresponds to each producer known by that consumer (issued independently because this is a decentralized system). Furthermore, each of these tags would have to look random and unique to avoid identifying the producers to third parties. Finally, we'd need some form of polynomial-time algorithm that can convert a polynomial size list of such tag-recognition keys into a function (e.g., something like a hash map) that maps tags to tag-recognition keys in time sublinear in the number of tag-recognition keys. We assume that no such algorithm can exist because it would have to exploit some pattern or structure in these necessarily random tags.

□

Appendix B

Spec

This section is simply an implementation recipe. You should *only* read this if you actually want to implement DRACL. If you just want to understand it, read the System Overview (chapter 2). If you just want to use it on your content-host, go back to the API (chapter 6).

B.1 Keys

First, we need to understand the different types of keys in this system:

Consumer/Producer Identity Key The consumer and producer both have asymmetric identity key pairs. These are plain-old PGP keys that are used by the consumer, producer, and AP but *not* the content host. If possible, these should be stored on a secure element.

AP Key Every AP has a standard SSL certificate issued by a trusted CA.

ACL Key This is a special asymmetric key (using our custom crypto) that consumer's use to access a producer's content. There are two parts: the ACL Public Key and the ACL Private Key. While producers issue individual ACL Private Keys to each of their friends, they issue a single ACL Public Key; otherwise, content hosts would be able to identify the consumer based on this public key. Again, as explained in the overview, there are actually two ACL Keys: one issued by the producer and one by the producer's AP

Producer Secret Every producer has a set of secrets used to generate ACLs and ACL Keys.

B.2 Datastructures

Here, we define the datastructures that DRACL uses. For convenience, we encode all messages and datastructures in CBOR. A more efficient encoding (e.g. Protocol

Buffers) could be used at a future stage.

Note: All numbers encoded as `bytes` are encoded in network order.

B.2.1 Common Data Structures

Below, we define a few common data structures that we'll need throughout the protocol.

First, we define a `type` field simply to make introspection easier.

B.2.1.1 `SignedEnvelope`

We define a `AuthenticatedEnvelope` type for encapsulating authenticated messages:

> `type "authenticated"`
>
> `data bytes`
>
> The authenticated data.
>
> `auth bytes`
>
> The signature/MAC.

When we use `authenticated(data)` in the following message definitions, we mean that the data "data" is wrapped in a signed envelope with signature/message authentication code "auth". Note: we don't specify the specific key, method, or type because *how* a message should be verified depends on the internals.

B.2.1.2 `EncryptedEnvelope`

We define a `EncryptedEnvelope` type for encrypting messages:

> `type "public_key"`
>
> `encrypt_type "pgp"`
>
> The encryption format. Currently, only `"pgp"` is supported.
>
> `data bytes`
>
> The encrypted data.

When we use `encrypted(enc_key, data)`, we mean that the data is first wrapped in a signed envelope, then an encrypted envelope. The key that's needed to decrypt `data` should be encoded in `data` (how this is done is up to the underlying encryption system).

B.2.1.3 `PublicKey`

We define a `PublicKey` type for describing public keys:

> `type "public_key"`
>
> `key_type "PKCS12"` (DER encoded), `"RSA"` (DER encoded RSA key), or `"PGP"`
>
> (binary format) Opaque key data (in the case of PKCS12, this contains the entire certificate chain).
>
> `key "key"`
>
> The actual key data.

B.2.1.4 `Name`

Finally, we define a `Name` type for identifying public keys. These allow indirection so that the actual underlying public key can change.

> `type "name"`
>
> `name_type "dns"`, `"pgp"`, or ”rsa”
>
> `dns` A domain name. An associated public key would be the full certificate chain (X.509) signed by a trusted root CA.

`pgp` A PGP key fingerprint.

`rsa` A DER encoded RSA public key.

`name bytes` The actual name.

B.2.2 `ACL`

Below is the specification for the internal format of an `ACL`.

`type "acl"`

`acl_key_authorities [Name]`

A set of names that map to the keys that need to have signed the ACL keys used to authenticate against this resource. That is, a consumer will need a non-expired ACL Key signed by each of the parties listed below.

In general, this will list the AP's domain and producer's RSA key (using the RSA format, not PGP).

For technical reasons, these can't be PGP names. Basically, we don't want to require PGP on the hosts.

`producer Name`

The producer's identity (using the identity system).

This is used by the consumer to learn the identity of the producer. This is also the key used to sign the ACL.

`producer_ap string`

The producer's AP (where to find the necessary keys).

`epoch u64`

An monotonically increasing integer used to prevent time travel. See the discussion for details.

`acl_v1 bytes`

The current ACL format. Versioned in case we decide to change the underlying crypto. See the crypto section (chapter 5) for more.

`user_data any`

Small opaque user-data blob (should be symmetrically encrypted by the producer). This allows the producer to record information about the ACL in the ACL itself (e.g., a description of the groups/users listed in the ACL).

`SignedACL` is defined to be an ACL wrapped in a signed envelope (signed by the producer).

B.2.3 ACL Keys

B.2.3.1 `ACLPrivateKey`

`type "acl_private_key"`

`public_key_fingerprint bytes`

A SHA256 hash of the associated ACLPublicKey. Not necessary but can't hurt.

`comment string`

A short comment written by the producer for the consumer. This is effectively a MOTD.

`epoch u64`

This key is only valid for ACL's with epochs less than or equal to `epoch`.

`secret_v1 bytes`

The actual secret key. See the crypto specification (§ B.3) for more.

B.2.3.2 `ACLPublicKey`

`type "acl_public_key"`

`signing_key PublicKey`

The public key that signed this ACL Key. Can either be an RSA key or a full PKCS12 key+certificate chain.

`expires u64`

Unix timestamp when this key expires.

In general, the producer's ACL Public Key will never expire (that's why we have an AP) but the AP's Public Key should expire on the order of a few hours.

Zero means never.

`public_v1 bytes`

The public part of the crypto protocol (see § B.3).

B.2.4 Authentication Protocol Data Structures

B.2.4.1 `Challenge`

`type "challenge"`

`crypto_challenge_v1 bytes`

The actual challenge as defined by the crypto protocol.

`user_data any`

Some extra data included by the content host. This will be returned along with the `ChallengeResponse` and is used to avoid having to keep state on the content host.

B.2.4.2 `ChallengeResponseBody`

The statement to HMAC as a Challenge Response.

`type "challenge\_response\_body"`

`origin string`

The server from which this request originated (authenticated by a secure channel such as TLS). This allows the server to detect MITM attacks.

`user_data any`

The extra data included in the challenge from this response was generated.

B.2.4.3 `ChallengeResponse`

The actual HMACed message.

`type "challenge\_response"`

`challenge_response_body bytes`

The body of the challenge response as bytes.

`hmac_v1 bytes`

The HMAC. See chapter 4.

B.3 Crypto Internals

Here we specify some of the cryptographic detail's we've omitted previously.

We use bn256 [14] as our pairing function, SHA256 as our HMAC function, and NaCl's[3] `secretbox` for symmetric encryption.

When transmitting cryptographic material, define the format to be a simple concatenation of all the (fix-sized) fields in the order in which they are described in chapter 5. In the future, it would be a good idea to define a precise format but this format must be extremely efficient.

An implementer should follow the protocol described in chapter 4 (using the crypto we've defined in chapter 5.