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 Advisor

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 practical, user friendly, secure, and *privacy-preserving* 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 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[1]).

Contents

Chapter 1

Introduction

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 their 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 with their 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 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 a one of a few "blessed" third-party accounts but these aren't standardized and doesn'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.

1.2 The Problem

We've identified three pieces needed to turn the web into a single social network: identity management, access control, and content syndication.

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

Do we include content hosting (e.g. solid)? I'm basically just taking it for granted that the "web" al-

(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.

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.

Content syndication addresses the problem of advertising and finding content. That is, while an access control system can permit or deny a consumer from accessing content, it doesn't actually tell the consumer where to find content they can access. In Facebook terms, this is the *feed*. To achieve our goal, we need a content syndication system that allows users to advertise discover content independent of where it is stored.

1.3 The Solution

Systems like Facebook Connect, Keybase, NameCoin, and GnuPG (OpenPGP) try to solve the identity management problem. 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, and none of the decentralized system have wide adoption. However, this *is* a well explored problem lacking only a simple, widely adopted solution.

We focus on solving the access control problem. We would like a practical, secure, privacy preserving, and developer and user friendly access control system for the web that allows people to manage who can access what across all the content hosts they choose to use through a centralized system.

We leave the content syndication problem for future work. The current most widely adopted solution is email. However, email is far from perfect..

We're still a long way off from a web without walls but we provide a large missing component.

David, citations please. This is your area.

1.4 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 published by a producer with the producer's permission.

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 party using this system 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 the standard cryptographic definition of honest but curious.

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

1.5 Requirements

We would like a practical, developer and user friendly, secure, and privacy preserving access control system for the web. 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 social networks from global adversaries. These are simply out of scope of an access control system and can be provided by a dedicated anonymization system.

1.5.1 Practical

By practical, we mean a good access control system should be designed to work in the world as it is, not as we wish it were.

For example, it 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.

Additionally, it can't assume that anyone can build a perfect implementation. No practical, complex system can hope to achieve perfect security and perfect uptime. At the end of the day, this system will be built and maintained by humans so we must design it with that in mind.

Finally, 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.

Practicality is a prerequisite for the success of any system. Therefore, an access control system that ever hopes to be more than words on paper must be practical (the worlds most obvious proof). Unfortunately, the number of clearly impractical systems that were never implemented indicates that this is less obvious than it should

be so it bears to be stated explicitly.

1.5.2 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, 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. Competition is not only good for users, it's absolutely necessary to keep

companies from screwing their users over in pursuit of more money. See AT&T[14] policy and Comcast's[15] recent attempt to charge an additional fee for not snooping on their customers browser history.

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.

In general, computers should do what they do best: repetitive tasks. A good access control system should abstract away much of the repetitive nature of socializing on the internet and let users focus on socializing (what humans do best).

1.5.3 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), it shouldn't require significant infrastructure changes, and it should make it easier (than no universal access control system) to bootstrap a new content host.

We believe that a lack of developer friendliness contributed significantly to the failure of systems like OpenID[3]. 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 [11].

Furthermore, developers should *want* to use this access control system because it can help developers avoid the bootstrap problem. In the web as-is, you need users to attract users because nobody wants to be alone on a content host. A good global access control system should help alleviate this problem by allowing users to access content on any content host without having to even so much as login.

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.

1.5.4 Secure

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

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.

This all comes back to practicality. It's not enough to be secure as long as nobody makes a mistake, a secure system must be able to tolerate mistakes and continue operating.

1.5.5 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 should not 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 be-

lieve 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.

Second, such a system should not reveal the 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 consumers explicit consent. Otherwise, the consumer loses the freedom to choose to either defer or avoid participation in a social interaction.

Third, such a system should 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 not to share a piece of content with a consumer because they don't want to be associated with the consumer. Consumers may choose not to 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.

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.

1.5.6 Summary

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

1. No single point of failure or bottleneck.

2. All compromises are recoverable.

3. Only permits explicitly authorized access.

4. Supports usable group management.

5. Requires consumers to manage at most one set of credentials.

6. Is easy to deploy on content hosts.

7. The producer's social network is private.

8. With whom a producer shares a piece of content is private.

9. The consumer should be able to access content anonymously.

1.6 Existing Systems

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

1.6.1 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 sys-

tems on top of the authentication system. In other words, there *is* no global access control system.

This type of system is inherently bad for privacy because content hosts learn the identities of who has access to what and, by extension, who knows who. While this privacy issue sufficiently motivates an alternative design, we have included this section to give an overview of some common existing systems and to learn from their shortcomings (in addition to the privacy problem).

1.6.1.1 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 [9].

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 (section 1.5.2), this is bad for usability.

From a developer usability standpoint, site-specific accounts force content hosts to implement custom account/access control systems and make it harder to attract new users because developers have to convince new users to "sign up" before they can participate. Again, refer the requirements (subsection 1.5.3) 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 [4] and users often reuse passwords and/or use weak passwords [17].

To be fare, site-specific accounts have served the web well for a very long time. This is primarily because they have no external dependencies and 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 better new solution.

1.6.1.2 Centralized Authentication

Centralized 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 password 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 authentication systems increase 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 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 which, again, violates our stated requirements. 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 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 [19] to make integration with content hosts easier, they must still be blessed on a one-by-one basis.

1.6.1.3 Decentralized Authentication

Decentralized authentication protocols such as OpenID [3], Persona [2], and WebID [8] allow users to identify to multiple services using the same credentials but, unlike centralized authentication systems, decentralized authentication protocols do not force users to choose between a few "accepted" providers. Additionally, in theory, content hosts should be able to support exactly one decentralized authentication protocol — assuming they eventually converge. While decentralized authentication protocols address the user choice concern, they still don't address our privacy con-

cerns because they still operate on the level of identity.

1.6.2 Access Control Systems

Unlike authentication systems, access control systems directly dictate what operations a system should and should not permit. Where an 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 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 by a unified system. Unfortunately, there aren't any existing access control systems that meet our privacy and security requirements.

1.6.2.1 Centralized Access Control

Centralized access control systems such as Kerberos [23] and LDAP [20] 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 centralization 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.

David, I added this last part in response to your comment but I really don't think this needs to be said. No service large enough to become a "blessed" access control service would want to interoperate with other services at any level as it would make it easier for users to jump ship.

1.6.2.2 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.

There are existing cryptographic protocols for decentralized access control such as Decentralized Access Control with Anonymous Authentication of Data Stored in Clouds [25] and Privacy-Aware Attribute-Based Encryption with User Accountability [21]. Unfortunately, these are simply cryptographic protocols and neither attempt to provide a full system designs. 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).

On the other hand, there are real decentralized access control system design such as DRBAC [18], Rule-Based Access Control for Social Networks (RBACSN) [16], and the Kerberos Consortium's User Managed Access (UMA) [7]. Unfortunately, none of these protocols even consider privacy; it simply doesn't factor into their designs. Furthermore, UMA and DRBAC both require their users to completely trust some AP with access to all their content. On the other hand, RBACSN doesn't allow restricting access to groups or individual consumers.

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.

1.7 Design

Now that we've determined that we need a decentralized access control protocol, this section explores the design space and works 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.*

1.7.1 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.

1.7.1.1 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 hence the.

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. 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 [12] fix these problems by issuing short-lived, non-url bearer credentials that can optionally require the bearer to prove knowledge of some additional key. Macaroons even support delegation. However, being short lived, we'd still need to design a system for issuing 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).

1.7.1.2 Encrypt All The Things

Another solution is to just encrypt all content on content hosts and only make it accessible to consumers that should have access.

This has *perfect* privacy. Nobody has to authenticate to anyone, no metadata is

leaked — other than how many consumers can access a piece of content. Basically, producers just encrypt the content to all recipients and upload it anonymously to a content host. Consumers download it and try decrypting it.

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 authorize *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.

1.7.2 Zeroth Attempt

The brain-dead solution is to have content hosts ask producers directly if 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.

1.7.3 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 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, this has has a some major drawbacks.

1.7.4 Don't Be A Weapon

For one, content hosts have to make network requests on behalf of clients (consumers) to servers (APs) specified by clients (producers). From a security standpoint, this 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[26] and/or wait for an untrusted party[24].

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) [7], operate this way.

1.7.5 Distribute The Storage

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 means that the AP's state will grow linearly with the amount of content published.

Our solution was to replace the CIDs with access control lists (ACLs) stored on the content hosts 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). Basically, we're just distributing the load by storing the ACL on the content hosts

instead of AP and using cryptography to make sure 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.

1.7.6 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 users of 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.

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, there's no way to expire a symmetric key. The only way to revoke a consumer's access to content would be to update the ACL. This violates our goal of being able to efficiently revoke access without updating every single ACL.

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 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.

1.7.7 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. Unfortunately, the CGBBs produced by this factory are 2x slower.

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 but you can read about that in chapter 2.

1.7.8 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).

1.8 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 (1.7) 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. In reality, this means 1000 "friends" because groups are the unit of access control. See the user friendliness section (3.4) for a 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. 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 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. 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.

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 provide a protocol for securely transitioning to a new key — authenticating the new key out of band — and restoring access.

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 key managed by teh assumed identity system.

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

ACL Key A key issued by a producer to a consumer for authenticating against ACLs.

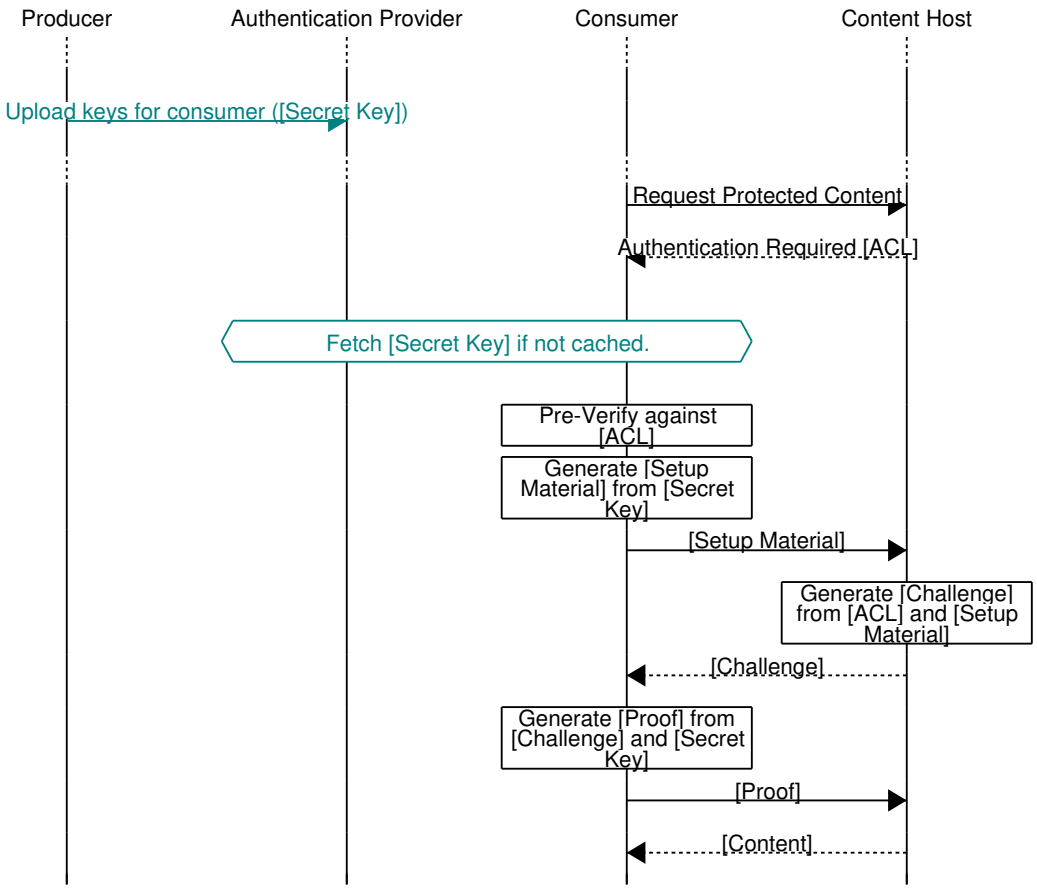
Producer Secret All the secret information needed for a producer to make their ACL keys.

2.3 Authentication

When attempting to access protected content, the content host sends the content's ACL to the consumer along with a challenge. The consumer then fetches their ACL Keys from the producer's AP if not already cached locally, pre-verifies that they will compute the correct challenge (by running the entire protocol locally), and computes

a challenge-response which it returns to the content host. The parts of this diagram that aren't strictly speaking part of the authentication protocol are marked with double parenthesis and covered later.

When attempting 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 determine if the consumer should be able to access the protected content. The parts of this diagram that aren't strictly speaking part of the authentication protocol are marked with double parenthesis and covered later.



After running this protocol, the content host learns precisely that the consumer has chosen to authenticate against an ACL and the consumer learns the identity of the producer (listed in the ACL) and how many groups grant them access to the resource (this is simply an artifact of the crypto).

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 three 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).

`authenticate` 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 this to prevent

malicious parties from learning the consumer's identity (see 3.2.2.1). To make this less of a pain, we expect browser 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 3.2.2.2).

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 Key with the consumer's identity key and signs it with its own identity key and uploads it to the AP along with the consumer's identity key. This is a slight simplification as the producer gives the AP a few extra pieces of information that we'll discuss in the next section.

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 privacy as this would necessarily require updating all old ACLs. 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 use a two-key system when authenticating against ACLs. Basically, every ACL Key is actually two keys: one issued by the producer that never expires and one issued by the producer's AP that expires quickly. When the consumer goes to the producer's AP to ask for their keys, they fetch both keys. When the consumer authenticates against an ACL, they use both keys. This allows the producer's AP to revoke access to any given consumer by simply not issuing its short-lived key. 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 (and a subkey of the identity key for implementation reasons).

Chapter 3

Discussion

Above all, DRACL is practical — that is, usable in practice, not only in theory. Additionally, DRACL is secure, privacy preserving, and developer and user friendly. This section evaluates how we stack up against our requirements for a "good" access control system for the web.

3.1 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 (yes, we know the current one has some issues) so all *security* compromises are recoverable as long as the CA system works.

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 contents current form (the AP can time-travel the social network, not content). A security conscientious 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.

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.

3.2 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. 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 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 3.2.1.1 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 and access their content.

Producer An honest but curious producer cannot learn if and when specific friends access their content. Given sufficient collusion and effort, producers can learn something about who accesses what when but this is unlikely to be an issue in practice. See section the on consumer privacy (3.2.2) for details.

The following sections detail what various privacy leaks we do have and why.

3.2.1 Producer Privacy

In general, DRACL reveals (almost) nothing about the producers social network to any third party other than the producer's AP.

3.2.1.1 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 know we can fix this by using stronger (more expensive) cryptographic primitives but can't 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.

3.2.1.2 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 provide guarantee 4 without sacrificing performance. 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 changes (requirement). This system must avoid contacting a third party such as the AP while authenticating (requirement). To achieve the first requirement, the ACLs must not fully specify the 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 pro-

ducer as the first simply because it will reuse the same producer specific information.

□

Providing guarantee 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 proven 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 provide guarantee 2, DRACL would either loose unidirectional friendships or scale horribly at the tail. Basically, consumers could remember a small list

of producers with which they are actually friends. For each producer, they could store a producer-specific 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 loose this property.

On the other hand, 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 a 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-specific secrets to find the producer that authored the ACL.

Above 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.

□

In short, we make the producer public because it allows us to achieve better usability and performance.

3.2.1.3 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 consumer's 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. Basically (massive simplification), the producer would get the AP to sign some keys where the consumer has been cryptographically hidden (blinded) but the producer is visible (to the AP). Then, the producer would unblind the part mentioning the consumer and blind the part mentioning the producer and re-upload the altered key to the AP over an anonymous connection. 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 blinded). The consumer would then have to unblind the part that mentions the producer to be able to successfully authenticate to content hosts.

Unfortunately, even the simplified this scheme above 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 uploaded a key for consumer B at time $T + \epsilon$, A is probably a friend of B.

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

3.2.1.4 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. However, we feel that, at this point, the chicken has flown the coop so to speak.

3.2.2 Consumer Privacy

DRACL allows consumers to anonymously access content. That is, they can, subject to some restrictions discussed below, authenticate and view content anonymously.

3.2.2.1 Doxing Consumers

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 can post a resource to a content host along with an ACL that lists a unique AP domain controlled by the attacker. When a consumer accesses the 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 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.

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.

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 3.2.1.3.

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 3.2.1.2.

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.

3.2.2.2 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.

3.2.2.3 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.

3.2.2.4 Side Channels

Finally, 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[6] or Tails[5].

3.3 Developer Friendly

To be developer friendly, DRACL wraps all content-host logic in a simple (interface wise), environment-agnostic library. 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 3.1 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.

3.4 User Friendly

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

3.4.1 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). DRACL 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 fill pages with how awesome this is (shared credentials is a huge 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 implemented in the browser itself and integrates deeply with the browser. Unlike systems like OpenID, Facebook Connect, and Google Sign-In, it doesn'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 the 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.

These last two issues have been cited [10] by Mozilla as reasons reasons their authentication system, Persona, failed.

3.4.2 Where We Could Improve

However, 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.

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 3.1 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.

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

3.5 Future Work

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

- Don't leak the number of groups that grant a user access to a resource.

- 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 (3.4.2).

  - 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 bit this may not be necessary.

  - We know how to cut the amount of crypto done by the client by a third however, this would make us rely on the secrecy of the Producer Secrets for security, in addition to privacy. It would also make the time travel problem worse and potentially introduce other unknown security problems.

  - We could also make 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. We could, with relative ease, allow granting access to some number of groups or (exclusive) an single consumer but this provides dubious benefit and complicates the underlying crypto.

– 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

Crypto

A huge thanks to Prof. Vinod Vaikuntanathan (CSAIL, MIT) for extensive help (and hand-holding) while developing the crypto for DRACL.

In DRACL, we needed a way for consumers to prove that the set of groups in which they are a member intersects the set of groups that listed in an ACL. There are three requirements that make this difficult:

1. These sets must remain secret from both parties.

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. Group membership must expire after a period of time.

4.1 Reduction To Math

To solve this problem with crypto, we first need to reduce it to math. In DRACL, every ACL includes an encrypted bit vector of groups $\vec{a}$ that can access the content and every consumer is given another encrypted bit vector of groups $\vec{b}$ in which they are a member (the ACL Key). When authenticating, the consumer takes the inner product of these two encrypted vectors, and, if the inner product is non-zero, the content host grants them access. The tricky part is (1) encrypting these vectors such that the consumer can prove to the content host that (2) the vectors have a non-zero inner product without revealing anything more than that and (3) that $\vec{b}$ has not expired.

To hide the vectors, DRACL uses a function-hiding inner product encryption scheme described by Bishop, Jain and Kowalczyk [13] along with the security extension described by Lin (in chapter 4) [22]. This allows us to compute an inner product of two secret vectors without ever revealing anything about the vectors other than the inner product.

The details of this inner product encryption scheme are a bit complicated so we're go-

ing to abstract them away. We define the functions `EncryptedInput`, `EncryptedInputS`, and `EncryptedResult`. The inner product encryption scheme can multiply (using a function `Pair`) $\texttt{EncryptedInput}_1(\alpha, \vec{a})$ and $\texttt{EncryptedInput}_2(\beta, \vec{b})$ to produce $\texttt{EncryptedResult}(\alpha, \beta, \langle \vec{a}, \vec{b} \rangle)$ where α and β are random parameters to the encryption scheme chosen when the vectors were encrypted. This scheme can also multiply encrypted scalars: $\texttt{EncryptedInputS}_1(\alpha, a)$ and $\texttt{EncryptedInputS}_2(\beta, b)$ to produce $\texttt{EncryptedResult}(\alpha, \beta, a \cdot b)$. These functions have the following properties:

1. Given $f(\textit{parameters} \ldots, a)$ and a scalar b, it's easy to compute $f(\textit{parameters} \ldots, a \cdot b)$, and $f(\textit{parameters} \ldots, a/b)$.
2. $f(\textit{parameters} \ldots, 0) = 1$.
3. `Pair` can only take parameters with subscripts that match the argument position. That is, one can call $\texttt{Pair}(f_1(\ldots), f_2(\ldots))$ but not, e.g., $\texttt{Pair}(f_1(\ldots), f_1(\ldots))$.

So, now that we can talk about this authentication protocol in terms of math, we can actually describe the protocol.

4.2 Authentication Protocol

In this section, we build up the authentication protocol step-by-step.

4.2.1 Authentication Protocol 1.0

In our first version of the protocol, producer includes an encrypted vector $C_1 = \texttt{EncryptedInput}_1(\alpha, \vec{a})$ and $C_2 = \texttt{EncryptedInputS}_1(\alpha, 1)$ in every ACL and an $K_1 = \texttt{EncryptedInput}_2(\beta, \vec{b})$ and $K_2 = \texttt{EncryptedInputS}_2(\beta, 1)$ in every ACL key. 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 $\langle \vec{a}, \vec{b} \rangle$ were zero. As we develop the protocol, this will get more complicated but this is a good starting point.

More concretely, the content host sends the consumer the ACL (C_1, C_2) and $\texttt{EncryptedInput}_1(\alpha, s\vec{a})$
The consumer then computes three things where c is the inner product of $\vec{a}$ and $\vec{b}$:

$$\begin{aligned}
\texttt{EncryptedResult}(\alpha, \beta, 1) &= \texttt{EncryptedResult}(\alpha, \beta, 1 \cdot 1) && (1)\\
&= \texttt{Pair}(\texttt{EncryptedInputS}_1(\alpha, 1), \texttt{EncryptedInputS}_2(\beta, 1))\\
&= \texttt{Pair}(C_2, K_2)\\
\texttt{EncryptedResult}(\alpha, \beta, c) &= \texttt{EncryptedResult}(\alpha, \beta, \langle \vec{a},\ \vec{b} \rangle) && (2)\\
&= \texttt{Pair}(\texttt{EncryptedInput}_1(\alpha, \vec{a}), \texttt{EncryptedInput}_2(\beta, \vec{b}))\\
&= \texttt{Pair}(C_1, K_1)\\
\texttt{EncryptedResult}(\alpha, \beta, s \cdot c) &= \texttt{EncryptedResult}(\alpha, \beta, \langle s \cdot \vec{a},\ \vec{b} \rangle) && (3)\\
&= \texttt{Pair}(\texttt{EncryptedInput}_1(\alpha, s \cdot \vec{a}), \texttt{EncryptedInput}_2(\beta, \vec{b}))\\
&= \texttt{Pair}(\texttt{EncryptedInput}_1(\alpha, s \cdot \vec{a}), K_2)
\end{aligned}$$

Given that c is small (less than 1000 in our case), the consumer can use (1) and (2) to brute-force c. They can then compute $\texttt{EncryptedResult}(\alpha, \beta, s)$ and send both this and K_2 to the content host. The content host can then compute $\texttt{Pair}(\texttt{EncryptedInputS}(\alpha, s \cdot 1)$, $\texttt{EncryptedInputS}_2(\beta, 1))$ and verify that the output is $\texttt{EncryptedResult}(\alpha, \beta, s)$.

There are two significant problems with this scheme:

1. The consumer hasn't proved to the content host that their key hasn't expired. That is, they've proved that they know some encrypted vector that has a non-zero inner product with the content host's vector but they haven't proved that they know one that's still valid.

2. β, and, in-turn, $\texttt{EncryptedInputS}_2(\beta, 1)$ and $\texttt{EncryptedResult}(\alpha, \beta, s)$ all uniquely identify the consumer.

We're going to fix (2) first because that will affect our fix for (1).

4.2.2 Anonymity

To hide the consumer's identity, we randomize the consumer's response to the content host in a way that still proves that that the inner product is non-zero. To do so, we literally just multiply the scalars in $\texttt{EncryptedResult}(\alpha, \beta, s)$ and $\texttt{EncryptedInputS}_2(\beta, 1)$ by some random parameter u to get $\texttt{EncryptedResult}(\alpha, \beta, u{\cdot}s)$ and $\texttt{EncryptedInputS}_2(\beta, u)$. Internally (pealing back the abstraction a bit), this is equivalent to permuting β.

4.2.3 Expiration

To support expiration, we would like to be able to write some certificate (using standard asymmetric cryptography) certifying an... TODO

Chapter 5

API

This chapter covers how to use DRACL as a content-host developer. This is probably the chapter you're looking for.

5.1 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 `authenticate(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.

5.1.1 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 3.2.2.3.*

The `response` argument to `callback` should be sent back to the server and run

through the server-side `authenticate` 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.

5.1.2 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. client When and if the callback executes, it should return `response` to the content host.
5. server On the content host, call `authenticate(secret, origin, acl, resp)`.

6. server If the result is `Deny`, abort.
7. server If the result is `Continue(challenge)`, send `challenge` back to the consumer's browser and go back to step 3.
8. server If the result is `Grant(until)`, grant access until the time specified by `until`.

Importantly, the server maintains no mutable state.

Chapter 6

Spec

Obi-Wan: These are not the specs you are looking for.

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 (chapter 2). If you just want to use it on your content-host, go back to the API chapter (chapter 5).

6.1 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.

6.2 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.

6.2.1 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.

6.2.1.1 `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 au-

thentication code "auth". Note: we don't specify the specific key, method, or type because *how* a message should be verified depends on the internals.

6.2.1.2 `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 crypto specification).

6.2.1.3 `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.

6.2.1.4 `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. TODO: Subkeys, key transitions...

`rsa` A DER encoded RSA public key.

`name bytes` The actual name.

6.2.2 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 {c1: [bytes], c2: bytes}`

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

`user_data bytes`

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).

6.2.3 ACL Keys

6.2.3.1 `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 {k1: bytes, k2a: bytes, k2b: bytes}`

The actual secret key. See the crypto section (**??**) for more.

6.2.3.2 `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.

6.2.4 Authentication Data Structures

TODO: Fix crypto protocol.

6.2.4.1 `Challenge`

`type "challenge"`

`acl SignedACL`

The ACL (signed).

`crypto_challenge_v1 [bytes]`

The actual challenge as defined by the crypto protocol.

`user_data bytes`

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.

6.2.4.2 `ChallengeResponse`

`type "challenge\_response"`

`crypto_response_v1` TODO

The actual challenge as defined by the crypto protocol.

`user_data bytes`

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

Chapter 7

Proofs

Gandalf: Fly, you fools!

TODO: Take the proof of security for the *old* crypto protocol and make it work with the new one (they're basically the same system repeated a bunch of times). This allows us to state that the security does not rely on the myriad of "producer secrets" we keep around.

7.1 Consumer Anonymity

In this section, we prove our anonymity requirement. We assume that the producer and authentication provider do not 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.

Whenever a consumer solves a challenge, they do two things:

1. They verify that the challenge was correctly formed.

2. They use a provably unlikable ACL Public Key.

7.2 Producer Privacy

In this section, we prove privacy. Specifically, we prove that DRACL hides the structure of its producers' social networks from third parties.

First, no party can directly learn which groups are listed in any given ACL or which groups a consumer is a member of. However, consumers can learn the size of the intersection between their groups and those listed in an ACL. This follows directly from the guarantee given by the underlying function-hiding inner product encryption scheme: the function-hiding inner product encryption scheme reveals nothing about

the "functions" (vectors) except their inner products (the size of the intersection).

Second, no third party learns what content a specific consumer can or cannot access.

...

7.3 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.

7.3.1 Assumption

To prove security, we assume that computing $\mathbf{e}(g_1, g_2)^{abc}$ is hard given:

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

$$g_2, g_2^a, g_2^b, g_2^c, g_2^{c^{-1}}$$

7.3.2 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.

7.3.2.1 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, let:

$$g_2^t = g_2^a \qquad g_1^{sq} = g_1^b$$

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

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) Let

$$C_2 = g_1^{\alpha \vec{d_1^*} + \alpha \vec{d_2^*}}$$

$$C_1 = (g_1^q)^{\vec{b^*}_{2n+3} + \vec{b^*}_{2n+1} + \sum_{\forall x_i \in \vec{x} | i \in \bar{\gamma}} x_i \vec{b^*}_i} (g_1)^{\sum_{\forall x_i \in \vec{x} | i \in \gamma} x_i \vec{b^*}_i}$$

6. Generate the target challenges:

$$_{rc}^{s} = \left(\prod (h_i^{sq})^{\tilde{\mathbf{A}}_{(\mathbf{r},\mathbf{k})} \mathbf{R}_{(\mathbf{k},\mathbf{i})}} \,\middle|\, i = 1 \ldots n \right) \qquad \forall r \in \tau$$

We can generate multiple challenges per resource. Is this necessary for the proof?

7. Generate keys for all epochs other than the current one. For each epoch, pick

t', for each user pick u:

$$
\begin{aligned}
{}_{ug} &= \left(g_0^{t'u},\, h_0^{u^{-1}}\right) \\
{}_{u} &= \left(\left(\prod_{k=1}^{n} \begin{cases} \left(g_i^{q^{-1}}\right)^{\mathbf{R^{-1}_{(i,k)}B_{(k,u)}}} & \text{if } k \in \bar{\gamma} \\ \left(g_i\right)^{\mathbf{R^{-1}_{(i,k)}B_{(k,u)}}} & \text{if } k \notin \bar{\gamma} \end{cases} \right)^{t'u} \middle| \, i = 1 \ldots n \right)
\end{aligned}
$$

8. Generate keys for the current epoch for all users not in groups $\bar{\gamma}$:

$$
\begin{aligned}
{}_{ug} &= \left(g_0^{t'u},\, h_0^{u^{-1}}\right) \\
{}_{u} &= \left(\left(\prod_{k=1}^{n} \left(g_i^{t}\right)^{\mathbf{R^{-1}_{(i,k)}B_{(k,u)}}} \right)^{u} \middle| \, i = 1 \ldots n \right)
\end{aligned}
$$

I had to get rid of β here. What's the correct way to do this?

Finally, pass all these variables to the function f yielding $\mathbf{e}(g,h)^{srtu\dot{c}}$. Given that we chose r and u and c is known to be a small number between 1 and n, we can then produce $\mathbf{e}(g,h)^{st}$ which is the same as $\mathbf{e}(g,h)^{abc}$. However, this violates the assumption so the function f cannot exist.

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

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

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

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

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

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

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

[10] Identity/persona aar, Mar 2014.

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

[12] 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).

[13] 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.

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

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

[16] 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.

[17] 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.

[18] 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.

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

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

[21] Li, J., Ren, K., Zhu, B., and Wan, Z. Privacy-aware attribute-based encryption with user accountability. In *Information Security*, P. Samarati, M. Yung, F. Martinelli, and C. Ardagna, Eds., vol. 5735 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2009, pp. 347–362.

[22] 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.

[23] Neuman, B., and Ts'o, T. Kerberos: an authentication service for computer

networks. *Communications Magazine, IEEE 32*, 9 (Sept 1994), 33–38.

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

[25] 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.

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