Title
Decentralized Authorization with Private Delegation

Author
Andersen, Michael P

Publication Date
2019

Peer reviewed|Thesis/dissertation

Decentralized Authorization with Private Delegation

by

Michael P Andersen

A dissertation submitted in partial satisfaction of the

requirements for the degree of

Doctor of Philosophy

in

Computer Science

in the

Graduate Division

of the

University of California, Berkeley

Committee in charge:

Professor David E. Culler, Chair
Associate Professor Deirdre Mulligan
Assistant Professor Raluca Ada Popa

Summer 2019

Decentralized Authorization with Private Delegation

Abstract

Decentralized Authorization with Private Delegation

by

Michael P Andersen

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David E. Culler, Chair

Authentication and authorization systems can be found in almost every software system, and consequently affects every aspect of our lives. Despite the variety in the software that relies on authorization, the authorization subsystem itself is almost universally architected following a common pattern with unfortunate characteristics.

The first of these is that there usually exists a set of centralized servers that hosts the set of users and their permissions. This results in a number of security threats, such as permitting the operator of the authorization system to view or even change the permission data for all users. Secondly, these systems do not permit federation across administrative domains, as there is no safe choice of system operator: any operator would have visibility and control in all administrative domains, which is unacceptable. Thirdly, these systems do not offer transitive delegation: when a user grants permission to another user, the permissions of the recipient are not predicated upon the permissions of the granter. This makes it very difficult to reason about permissions as the complexity of the system grows, especially in the federation across domains case where no party can have absolute visibility into all permissions.

Whilst several other systems, such as financial systems (e.g. blockchains) and communication systems (e.g. Signal / WhatsApp) have recently been reinvented to incorporate decentralization and privacy, there has been little attention paid to improving the authorization systems. This work aims to address that by asking the question "How can we construct an authorization system that supports first-class transitive delegation across administrative domains without trusting a central authority or compromising on privacy?"

We survey several models for authorization and find that Graph Based Authorization, where principals are vertices in a graph and delegation between principals are edges in the graph, is capable of capturing transitive delegation as a first class primitive, whilst also retaining compatibility with existing techniques such as Discretionary Access Control or Role Based Access Control. A proof of permission in the Graph Based Authorization model is represented by a path through the graph formed from the concatenation of individual edges. Whilst prior implementations of Graph Based Authorization do not meet the decentralization or privacy-preserving goals, we find that this is not intrinsic, and can be remedied by introducing two new techniques. The first is the con-

struction of a global storage tier that cryptographically proves its integrity, and the second is an encryption technique that preserves the privacy of attestations in global storage.

The horizontally-scalable storage tier is based on a new data structure, the Unequivocable Log Derived Map, which is composed of three Merkle trees. Consistency proofs over these trees allow a server to prove that objects exist or do not exist within storage, as well as proving that the storage is append-only (no previously inserted objects have been removed). Our scheme advances prior work in this field by permitting efficient auditing that scales with the number of *additions* to the storage rather than scaling with the *total* number of stored objects. By utilizing cryptographic proofs of integrity, we force storage servers to either behave honestly, or become detected as compromised. Thus, even though the architecture is centralized for availability and performance, it is does not introduce any central authorities.

The design of the storage does not ensure the privacy of the permission data stored within it. We address this through the introduction of Reverse Discoverable Encryption. This technique uses the objects representing grants of permission as a key dissemination channel, thus operating without communication between participants. By using Wildcard Key Derivation Identity Based Encryption in a non-standard way (with no central Private Key Generator) we allow for permission objects to be encrypted using the authorization policy as a key. Thus, RDE permits the recipient of some permissions to decrypt other *compatible* permissions granted to the grantee that could be concatenated together to form a valid proof. RDE therefore protects the privacy of permission objects in storage whilst still permitting decryption of those objects by authorized parties.

We construct an implementation of these techniques, named WAVE, and evaluate its performance. We find that WAVE has similar performance to the widely used OAuth system and performs better than the equally widely used LDAP system, despite offering significantly better security properties. We present an advancement to Graph Based Authorization which efficiently represents complex authorization proofs as a compact subgraph rather than a sequence of linear paths, and present a technique for efficient discovery of such proofs.

To validate our techniques and ensure their efficacy in practice, we pose an additional question: "How can we leverage WAVE to improve the security of IoT communications?" We present a microservice architecture that abstracts the interfaces of IoT devices to permit a uniform security policy to be applied to heterogeneous devices of similar function. This is achieved by enforcing security policy at the communication bus and using hardware abstraction microservices to adapt the interfaces that devices expose on this communication bus. We construct and evaluate an instance of this communication bus, WAVEMQ and find that, with appropriate caching, its performance is comparable to that of prior publish/subscribe information busses. We discover that by enforcing WAVE's security model in the core of the network, we gain a resistance to denial of service attacks. This is particularly valuable in the IoT context where devices are typically resource constrained or connected by a bandwidth-limited link.

Dedication

To my soon-to-be wife, Soo Hyun Kim, who supported me throughout my PhD, and to my family, friends, and advisors, who made this possible.

Contents

List of Figures

List of Tables

Acknowledgments

I would like to thank the Fulbright Program for giving me a scholarship to study at UC Berkeley. I would not be where I am today if it were not for them, and it has been an amazing journey.

Funding for my research was provided by Intel/NSF CPS-Security #1505773 and #20153754, DoE #DE-EE000768, NSF CISE Expeditions #CCF-1730628, NSF GRFP #DGE-1752814, and the supporters of the RISE lab: the Sloan Foundation, Hellman Fellows Fund, Alibaba, Amazon, Ant Financial, Arm, Capital One, Ericsson, Facebook, Google, Intel, Microsoft, Scotiabank, Splunk and VMware.

I would like to thank my advisor, David E. Culler, who has guided me through every step of my PhD, from the day he found my application to UC Berkeley, to the day he hooded me at my graduation. His insights and mentoring have had a profound impact on both my research and me personally. I am especially grateful for his respect of my personal research goals, where I was given the freedom to explore the problems I found interesting, rather than being forced into the research agenda of a lab or funding agency.

Thank you to Raluca Ada Popa, who joined my committee as co-advisor when my research began to delve into the security aspects of IoT. Her experience in this area was invaluable. Raluca was always available to give advice or lend a hand, even in the final hours before paper deadlines, and I am grateful for her assistance.

Thank you to Deirdre Mulligan, who joined my committee to offer an outside perspective on the societal implications of rearranging systems by introducing cryptography. Her advice has helped me analyze and express the consequences of using WAVE.

Throughout the course of my research, I was fortunate enough to work with many talented computer scientists who both contributed to my research and validated it through deployments. I would especially like to thank Sam Kumar, who was immensely helpful throughout my PhD. His highly efficient implementations of the core cryptography in WAVE were essential to its usability. Thank you also to the SDB/BETS lab: Gabe Fierro, Jack Kolb, Kaifei Chen, Moustafa Abdelbaky, Hyung-Sin Kim, and Kalyanaraman Shankari who all played a big part in my research, offering advice and incorporating pieces of my research into theirs. Their feedback helped guide WAVE to where it is today.

Thank you to my parents, Ingrid and Walter. You helped instill the curiosity within me that drove me to research. Thank you for your support over throughout my life, and for the sacrifices you made to get me where I am. Thank you, Ingrid, for keeping me on track through the rough patches in my PhD.

Thank you to the staff who helped keep things moving over the years. I am especially grateful to Albert Goto who ensured that I had everything I needed for my research. There wasn't a problem that Albert couldn't solve, and he worked tirelessly through many nights so we could make deadlines.

Finally, I would like to thank my fiance Soo Hyun Kim for her patience and support over the past few years. I could not have done this without her.

Collaborators

Large portions of this work are published in [7]. Portions of this dissertation are joint work with others. The optimized implementations of the cryptography used in Chapter 5 are the work of Sam Kumar, more fully described in [78]. The XBOS system in Chapter 7 is joint work with Gabe Fierro and others, more fully described in [6]. The Spawnpoint system in §7.2 is the work of John Kolb and others, more fully described in [6, 76].

Chapter 1

Introduction

1.1 Authentication and Authorization

Whenever there is an interaction between some kind of client and some kind of service, be it a person interacting with a web service or an app interacting with a device in a smart home, there are always two questions that must be answered before the parties can engage in digital dialogue. The first is "who is the client?". Answering this question is called *authentication*. The next question is "what is that client permitted to do?", which is *authorization*.

These questions are among the most pervasive in computer system design and can be found in nearly every digital system across every aspect of our lives. We wake up and authenticate with our social networking sites to see what happened overnight. We turn on our smart lights using our phone that has been paired with our home. We use a ride-sharing application to get to work, using a persistent session maintained by an authentication cookie on our device. We tip the driver using a credit card that we have proven we are authorized to use by reciting its number. We begin our work day by signing in to our email using a username and password. We authenticate with an online collaborative document editing suite, reviewing documents that our colleagues have given us permission to view. When we finally get home, we sign in to entertainment systems, like Netflix and Steam, using a combination of session tokens on our devices and, periodically, usernames and passwords. It is hard to find an aspect of our life that does not in some way or another involve an interaction with an authentication and authorization system and the exchange of some key that unlocks a piece of our virtual or physical world.

Despite the massive variety in systems having an authentication and authorization component to them, they are generally implemented with a common architecture, as shown in Fig. 1.1. The client sends credentials to a server that compares them to a hash stored in a database. If they match, it looks up an authorization policy for the user and applies to the session. In some cases, especially when authenticating services instead of people, a pattern like Fig. 1.2 is used where the client contacts the authorization server directly, obtaining a token that proves both authentication and authorization, which it sends to the service provider. This architecture is not very different, however: both of them have a logically centralized authorization server that establishes identity

Figure 1.1: The common authorization architecture where a service retrieves authorization policy from a centralized server

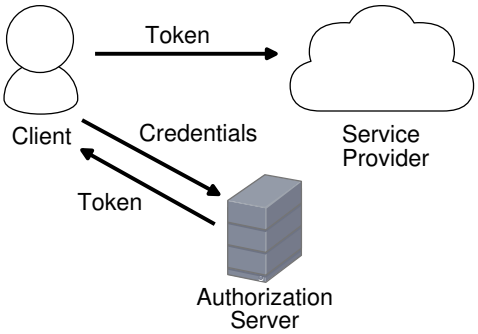


Figure 1.2: A bearer-token style authorization architecture where the client communicates with the authorization server directly

and authorization policy for an interaction with the service provider.

In some cases the centralization in these patterns spans authorization domains; it is common for websites to provide single sign-on in the form of a "Sign in with Google" or "Sign in with Facebook". In these cases, there are thousands of distinct companies all relying on a single centralized authentication service. In addition, the security of all of these systems usually relies on TLS and the Certificate Authority infrastructure, which is centralized.

This architecture is ubiquitous, used in hundreds of thousands of systems that we rely on in countless ways. Failures in these systems have the capacity to ruin people's lives and result in the loss of millions of dollars. Unfortunately, we find that this architecture has three critical problems, detailed in the next three sections.

1.2 Centralization

Existing mainstream authorization systems are typically built around a central database that stores users and their associated permission data. This approach presents two fundamental problems. First, a centralized service is a central point of weakness: a single attack can simultaneously compromise many user accounts and permissions. There have been numerous such breaches [80], where the compromise of the authorization server allowed attackers to log in as arbitrary users and perform actions on their behalf. Second, the operator of the central server has a complete view of

the private permission data for all users. Permission data contains sensitive metadata which, for example, can betray social relationships [104]. In addition to viewing permissions, operators can modify permissions without user consent [52], resulting in violations of user privacy.

Responding to the weaknesses of centralized systems, recent security systems are increasingly avoiding a trusted central service. This approach has been adopted by end-to-end encrypted communication systems [46], such as WhatsApp and Signal, blockchains (e.g., Bitcoin, Ethereum, Zcash), or ledgers (e.g., IBM's Hyperledger [28], Certificate Transparency [82],
Key Transparency [64]). Additionally, we have seen a paradigm shift in the engineering of secure distributed systems; Google argues in their BeyondCorp series of whitepapers [111, 92, 103, 19, 47, 74] that it is no longer possible build a secure network by enforcing security policy at the perimeter. Within a large network, it is necessary to treat every server within that network as potentially compromised. The whitepapers recommend that large corporate networks are constructed to have services operate with a degree of isolation. The compromise of any service is expected, and as a result does not result in the compromise of every other service, in contrast with the trusted-internal-network model.

Despite this recognition of the dangers of centralized security systems, we have seen little movement away from centralized authorization systems in practice. This is unfortunate, because the authorization infrastructure stands to benefit the most from the BeyondCorp mentality. Authorization is the cornerstone of system security: all services rely on the authorization infrastructure. If the authorization infrastructure itself is compromised, this cascades throughout the network and results in the compromise of most other services. An attacker in control of a centralized authorization server can manipulate arbitrary permissions on any user and change their password. Compromised authorization infrastructure, therefore, acts as a beachhead from which attackers can compromise other services by masquerading as valid users with valid permissions. This is difficult for a service to protect against because the traffic would appear like it was an authorized request from a valid user.

The logical response to the severe vulnerabilities of centralized authorization architectures is to investigate the construction of a system with no centralized authorities that is not vulnerable to such attacks and where end users can control who has access to their assets without relying on an authorization authority to do so on their behalf.

1.3 Federation Across Administrative Domains

The requirements placed upon the authorization infrastructure become more complex when you consider federation across multiple administrative domains. With traditional authorization systems, the operator has absolute authority over the authorization; they can view all the authorization policy, alter the policy or delete the policy. The owner of a resource must ask the authorization infrastructure to make changes when they wish to share access. This is often desired functionality in a simple setting because all participants defer to a single root of trust that can manage all the users and permissions stored on the system in a straightforward manner.

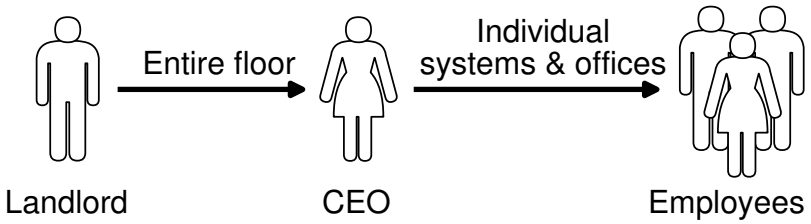


Figure 1.3: Authorization spanning administrative domains where a company rents a floor of a building

In a multiple authorization domain scenario, this model does not work. Consider the scenario depicted in Fig. 1.3. There is a landlord who owns a building and has authority over all the cyber-physical resources within that building. A small startup signs a lease to rent a floor of this building from the landlord. Consequently, the landlord must grant permissions for building assets within that floor to the CEO of the startup, giving him/her control over the HVAC, lighting and access control sytems on that floor. In turn, the CEO must delegate portions of that control to employees within the company. Perhaps one person is in charge of physical access control, so must receive permissions on the card key system. Perhaps another is in charge of facilities, such as HVAC and lighting. Employees with offices may want permission to modify the temperature set points within their office. In essence, there is a whole set of permission grants that occur within the company.

If we try manage this using a traditional authorization system, we find ourselves having to answer the question "who runs the server?". Either the landlord, the tenant, or a third party would need to run the authorization infrastructure that controls access to the building assets. None of these options are acceptable, however. We have two authorization domains: the landlord's, which contains the relationship with the tenant, and the company's which contains the grants within the company. Ideally, there would not be a party with visibility into and control of permissions that exist outside its own authorization domain but there is no way to achieve this using a centralized authorization server. If the landlord runs the authorization provider, then they can see all of the delegations that the tenant performs, which reveals internal corporate relationships to the landlord that might be renting to competitors. This is a violation of the company's privacy. If the tenant runs the provider, they are in a position to alter the permissions of the landlord, or their own permissions, which is an escalation of privilege. If a third party runs the service, then that third party has both complete control of and complete visibility into the permissions granted by the landlord and tenants. This means that the third party needs to be trusted absolutely, and becomes an external liability: compromise of the third party results in the compromise of all of their clients intimate authorization data.

Any traditional centralized authorization system requires one or more participants to trust an operating party to maintain the integrity and privacy of the authorization information. This is not ideal, and there are clearly benefits to solving federated authorization without this compromise.

1.4 Transitive Delegation

Systems today provide a multitude of functionality that users would want to be able to share with other users, and this is only becoming more true as every aspect of our lives becomes more and more reliant on on-line services. For example, a user interacting with a document editing suite would naturally want to be able to share the ability to read and edit some of those documents with other users. In the IoT case, the owner of a house may want to share the ability to control the lights and thermostat with other occupants of the house.

Delegation is supported in varying degrees by different systems, ranging from present in a crude form to completely absent. To extend the examples above, Google Docs or Dropbox allow you to share a document with another user, delegating the ability to read or write it. In contrast LiFX smart light bulbs and Netflix do not have delegation capabilities and require you to share your username and password with someone if you wish them to have any permissions at all. The latter case is obviously bad: the absence of delegation results in users sharing complete access to the account including, for example, billing details, which is a security risk.

What is less obvious is that the form of delegation in Google Docs, Dropbox, and most systems that *do* offer delegation, is also flawed. These systems almost universally use delegation with no provenance and transitivity. Transitive delegation means that if a person `A` grants permissions to a person `B` and that person further delegates to a person `C`, then the permissions that `C` has are predicated upon the permissions that `B` has. Consequently, `C`'s permissions can never exceed that of `B` and if `B`'s permissions are revoked, then so are `C`'s. In Google Docs, when `B` shares the document with `C`, an entry in the central ACL is created that is no different than if `A` had granted `C` the permissions: it does not capture how `C` obtained permissions. To make matters worse, `B` is not even capable of further delegating to `C` unless they have "admin" rights which allows them to remove the permissions that `A` has. In this paradigm, if `B`'s permissions are removed then it has absolutely no effect on the permissions of `C`. This presents two problems: firstly, it is unintuitive. If `B`'s permissions are being revoked, that is an indication that `B` is no longer trusted and therefore there is no reason that the people that `B` trusts (such as `C`) should retain permissions. It is impossible for `A` to reason about delegations that `B` might have made when this is not captured anywhere. Secondly, lack of transitive delegation allows `B` to perform a Sybil attack where it grants permissions to another account (say `D`) that it has created to mask its identity. When `B`'s permissions are revoked, it still has access through `D` and `A` has no idea that `D` was granted permissions from `B` and should therefore not be trusted.

In the example presented in the federated authorization case above, where a landlord has given permissions on a floor of a building to a tenant, you can see the importance of transitive delegation. In a secure system, the landlord would have no visibility into the permissions made by the CEO to the employees within the company. Despite this, if the landlord cancels the tenant's lease, all of the delegations made within the company should also be invalidated. It is intuitive, and necessary, for the permissions granted within the company to be predicated upon the permissions of the company as a whole.

In general, transitive delegation makes it easier to manage permissions at scale and results in a more secure system, especially in the face of limited visibility as delegations cross administrative

domains. As an example, the TLS Certificate Authority structure, which is one of the most widely used security systems ever created, has transitive delegation. It spans multiple administrative domains, from a handful of top level domain authorities, through to hundreds of millions of second level domains owned by companies or individuals. The validity of a certificate is predicated upon the validity of the certificate that signed it. Thus, the revocation of a certificate authority invalidates all certificates issued by that authority, so that even though the certificate authority is not aware of all delegations "downstream", they are appropriately and intuitively invalidated upon revocation of the CA.

At present, authorization at scale, such as within a large company, is predominantly managed as a single large access control list on a directory server that stores only the result of delegations rather than storing authorization policy in a form that mirrors the actual trust relationships that led to the permission grants. This monolithic, single-level access control list approach does not permit re-evaluation of permissions upon changes to the underlying trust relationships. We discuss in Chapter 3 how one can represent authorization in a form that captures trust relationships and permits transitive delegation.

1.5 Motivation and Thesis Question

The questions this work seeks to answer are twofold. How can we construct an authorization system that supports first-class transitive delegation across administrative domains without trusting a central authority or compromising on privacy? Further, how can we leverage such a system to improve the security of IoT communications?

There are several parts to these questions:

First-class transitive delegation is both the inclusion of delegation, an ability that is demonstrably necessary across most on-line services, and transitivity, a property that improves the manageability of authorization policy by allowing an action, such as the revocation of permissions, to have intuitive effects. In the absence of transitive delegation, a centralized authority with the capability to audit all the permissions becomes necessary as there is no other way to meaningfully reason about who is authorized.

Cross-administrative-domain functionality is necessary as systems increasingly span organizations. Examples within the built environment are abundant, as discussed above, but this also appears within networks in the BeyondCorp paradigm where individual services are modelled as isolated administrative domains to maintain privacy and integrity in the face of system compromise. Unified authorization policy spanning services therefore appears as crossing administrative domains.

No central authorities means avoiding the introduction of a party with access to sensitive authorization information or the ability to modify authorization policy. By avoiding the creation of such a party we reduce the attack surface of the entire distributed system by removing what is commonly a cross-cutting vulnerability.

Without compromising on privacy means that the mechanisms used for removing central authorities and permitting transitive delegation across administrative domains must not offer a level of privacy below that of existing mainstream authorization systems. This constraint was introduced after it became apparent that many obvious methods for meeting the other goals would have unacceptable privacy characteristics.

Improving the security of IoT communications means exploring the value of a decentralized authorization system in the context of IoT and the built environment where there exists many motivating use cases. It serves as a grounding example that ensures that the authorization system is capable of handling the authorization demands of real systems, with modest capacity, while also meaningfully contributing to the security of an emerging class of services and devices that are well known for having less than ideal security characteristics.

We are motivated to answer these questions through the engineering of a working decentralized delegable authorization system as we believe that the societal-scale impacts of centralized authorization systems, on both a security and user privacy front, demand immediate attention. We hope that an existential proof that decentralized delegable authorization is possible will spur industry into a shift away from the existing patterns of restriction and centralized authorization.

1.6 Solution Overview

The development of a decentralized authorization system was originally motivated by research into building operating systems. These systems frequently require interactions across domains of authority, so this drove us to discover a solution for federated authorization. The built environment served as a sanity check throughout development: at each stage we were able to evaluate the efficacy and usability of the authorization system against concrete use-cases.

This motivating suite of use-cases also led us to develop software that builds upon the decentralized authorization to solve other problems arising in distributed systems that span domains of authority. This section gives an overview of the entire solution, including these other parts. We begin with a description of the use-case.

The built environment consists of multiple buildings owned by different people. Within each building there is a collection of devices that operate the building, such as air handling units, variable air-volume boxes and thermostats. The collection of devices differs between larger buildings and, say, residential buildings where you are more likely to find commodity IoT devices like smart light bulbs or voice assistants, but the general problem is the same across the spectrum of buildings. There are devices that perform control (actuators) and devices that sense (e.g. air temperature sensors). There is also a collection of services that interact with the devices in the building by consuming sensor data and controlling actuators. Some example services include schedulers that turn down the heating when the building is expected to be unoccupied or demand response agents that react to signals from the electric utility to reduce peak power consumption.

In parallel with the devices and services in the building is a collection of people that live or work in the building and need to interact with it. Different people are permitted different levels of control over the building: a visitor might be able to turn on the lights but is probably not allowed

to adjust the thermostat. A manager might be free to adjust the thermostat but not interact with settings on the air handler that require expert knowledge to configure correctly. Regardless of how these permissions are enforced, the flow of permissions is typically hierarchical. At the top you would have the building owner who designates people he or she trusts to interact with the building, such as the tenant who signed the lease or a facilities manager hired to ensure everything runs smoothly. The facilities manager then lets other people exercise a subset of that control, such as letting department heads adjust the lighting, thermostat set points or door locks and those people in turn might delegate subsets of those permissions to others, such as support staff. This can be visualized as a graph, where permission flows from the owner, down through levels of the organizational hierarchy, to the occupants who work in the space. In many cases, some of the people in that graph may never exercise their permissions themselves: the department head will probably never adjust the heating schedule, but they are the person who decides who should have that responsibility, so the permissions still logically flow through them.

Typically, permissions for devices, especially commodity IoT devices such as smart thermostats, are managed by deciding who has the username and password that has been configured on that device. This does not map cleanly on to the natural flow of permissions through the organizational hierarchy. For example, if member of the support staff leaves the company, there is no way to remove just their access from the device without affecting everyone else who has access. Much of the complexity associated with securing large collections of smart devices and their associated services comes from a mismatch between how permissions are handled between people (the flow of permissions through the organizational hierarchy), and how devices handle their permissions using multiple independent access control lists.

WAVE is a distributed system we designed to represent and interact with permissions in a manner that mirrors the flow of permissions between people, thereby reducing the complexity of management while also improving the security. The people, devices and services in an *authorization graph* of permissions are digitally represented by *entities*. The edges in the graph capture the trust relationships between people and the permissions that have been delegated, are represented by *attestations*. We discuss the authorization graph in Chapter 3.

To fully capture the permissions associated with a heterogeneous set of devices and services within the built environment, it is necessary to have a unifying abstraction. Our building operating system work chooses to place this abstraction at the communication layer. All devices and services communicate over a publish/subscribe information bus, provided by WAVEMQ, discussed in Chapter 8. Messages are associated with a *resource*, to which authorized parties can publish or subscribe. For example, a thermostat would subscribe to a resource representing its setpoint, and a controller would periodically publish adjustments of the setpoint to that resource.

Resources exist within a *namespace* associated with a namespace entity which represents the domain of authority. A namespace entity might be created for a company, or a building, for example. Although resources are grouped into namespaces, they can be interacted with across multiple domains of authority. As an example, demand response control signals from the electric utility are subscribed to by controllers in multiple buildings, but the resources are located in the utility's namespace. A full resource, placed within a namespace, is represented by a URI, such as `namespace/service/interface`.

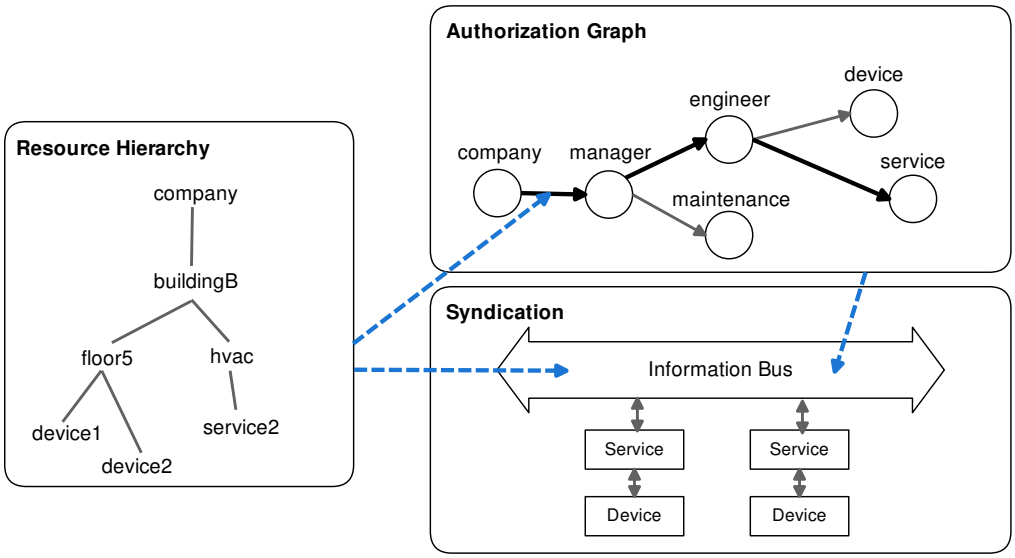


Figure 1.4: A high level view of WAVE integrated with syndication

The namespace entity "owns" all the resources within the namespace and intrinsically has permissions to interact with any of the resources within that namespace. The namespace entity can delegate permission to interact with a set of resources to another entity, and that entity can transitively delegate portions of that further. Any entity that interacts with a resource must have received permissions directly or indirectly from the namespace authority by the time the interaction occurs. These delegations are present as attestation edges in the authorization graph. A *proving* entity, such as one about to interact with a resource, can show that they have permission to interact with the resource by showing the path in the graph that begins at the namespace entity and ends at itself. Every edge in the path must grant a superset of the permissions that the proving entity is trying to exercise. We call such a path through the graph, composed of attestations, a *proof* as it is an independently verifiable proof of permission.

A proving entity constructs a proof by beginning at the namespace entity and performing a search through the graph for a path that terminates at the proving entity where every traversed edge grants a superset of the required permissions. A proof is verified by checking that the constituent attestations are a valid path (the creator of each attestation is the recipient of the previous attestation) and that the intersection of all the permissions granted by the attestations is a superset of the permissions claimed by the proving entity.

A key principle in WAVE, that every action should be accompanied by a proof that the action is authorized, is intrinsic to communication using WAVEMQ. Every message that is sent by a device, service or person contains a proof in the form of a path through the graph that is verified by both the communication infrastructure (the software routing the messages) and the recipients of the message (those entities that are subscribed to the resource the message is published to). For example, say there are temperature sensors in a room, and a thermostat that is connected to

a roof top unit. The temperature sensors and the thermostat would have services associated with them that interact with WAVE and WAVEMQ. The temperature device service would publish the temperature readings on a resource and the thermostat device service would listen for actuation messages on a resource. A controller service that wishes to bind the temperature sensors and the thermostat would subscribe to the temperature sensor resources and publish to the thermostat actuation resources. When subscribing, the controller service would need to construct a proof showing that it is permitted to read the resources associated with the temperature sensors. When publishing, it would need to construct a proof showing that it can publish to the thermostat actuation resources. All of these resources would be in the building namespace.

Figure 1.4 shows how these pieces fit together. Resource hierarchies in the form of URIs are present in the edges in the authorization graph and are also used in the information bus as rendezvous points where messages can be published and subscribed to. The information bus validates the portions of the authorization graph submitted by services as proofs that their messages are authorized. When messages are delivered to devices, they re-validate the proof to ensure that the bus itself has not tampered with or forged the message.

WAVE has been designed and implemented iteratively. There have been three generations of WAVE, with each offering different functionality. This is especially true of the mechanism used to store the authorization graph, described in Chapter 4. There we discuss the different versions of the storage tier and the lessons learned from each.

1.7 Generalization

The above frames WAVE in the context of buildings where resources are associated with the devices and services that comprise the built environment. These resources correspond to message delivery queues in the information bus provided by WAVEMQ. While this formulation was a guiding use-case in the development of WAVE, the authorization model is generally applicable to other scenarios.

Most common patterns for authorization can be mapped onto a resource-oriented authorization graph model of authorization. For example, Discretionary Access Control (DAC), such as that used in most social media sites, file sharing sites and file systems can be implemented with WAVE very cleanly. The resources in WAVE policies map onto the resources in the DAC ACL tables, and the permissions correspond to the permissions in the ACL tables. The advantage is that a WAVE proof can be validated by the individual servers handling the incoming request, rather than verifying the authorization at a central server that represents a single point of weakness, whose compromise would cascade throughout the whole system. The notion of limiting delegation can also be captured by WAVE, where an entity may share a picture with a group but not permit further sharing or delegation by members of that group.

In addition, WAVE can represent various formulations of Role Based Access Control (RBAC). Together, DAC and RBAC represent the vast majority of authorization patterns in use today. We discuss these further in §3.7.

There are also authorization tools that work with multiple patterns, such as OAuth [91] where a token resolves to a set of *scopes* that can be interpreted by an application as either DAC or RBAC policy. A WAVE proof can be considered a replacement for an OAuth token, resolving to a set of resources and permissions, rather than scopes. These can be interpreted, like in OAuth, as either DAC or RBAC policy. The advantage of WAVE over something like OAuth is that it removes the central token issuing server: an entity generates the proof by itself and can provide it to other services directly without going through an intermediary.

This general applicability of WAVE means that it can be used in point-to-point communication in addition to the publish/subscribe architecture detailed above. In fact, the publish/subscribe architecture can be thought of as a collection of point-to-point links decoupled by queues. WAVE can be applied to popular frameworks for point-to-point communication, such as HTTP REST APIs or GRPC APIs.

As the proof is standalone (meaning the verifier does not need to communicate with the prover or some third party server), the graph based authorization pattern can even be applied to asynchronous communication methods, such as email. A recipient of an email can verify that the sender is authorized to communicate with a given distribution list, or authorized to discuss certain topics represented by resources.

1.8 Roadmap

The dissertation is laid out as follows:

Chapter 2 discusses the prior work in this area and examines why we believe these systems have failed to garner widespread adoption. In so doing we establish a set of goals and constraints that we believe a solution system would have. Notably, a solution must provide transitive delegation without reliance on central trust. It must allow delegation with no constraints on ordering, operate with offline participants and allow for anyone to verify authorization. While meeting these goals, it must keep permissions private.

Chapter 3 introduces Graph Based Authorization, a model capable of meeting the requirements established in Chapter 2. GBA stores authorization as a graph where edges are delegations of permission and vertices are participants in the system. Proof of authorization is a path through that graph. The discussion of GBA surfaces the need for a place to store the graph.

Chapter 4 discusses the three generations of such a storage tier that have been used for storing the global authorization graph during the development of WAVE. The final tier is based on an Unequivocable Log Derived Map, a new data structure that can cryptographically prove its integrity, thereby forcing the operator of the storage server to behave honestly. The design of the storage tier makes all attestations public, which necessitates a mechanism for protecting privacy.

Chapter 5 introduces Reverse Discoverable Encryption which protects the privacy of attestations by encrypting them before they are placed in storage. The protocol uses attestations for key dissemination, so that entities can decrypt attestations that they need to use in a proof. RDE meets the requirements in Chapter 2: attestations can be granted in any order and decryption can occur without communication between entities.

The design and evaluation of an implementation of these techniques, WAVE, is discussed in Chapter 6. We show that WAVE has performance comparable to existing authorization systems despite offering better security guarantees. We also discuss advanced proof building that represents permissions as a subgraph rather than a linear path.

To answer the second thesis question, regarding improved security in IoT, Chapter 7 presents a microservice architecture for IoT and the built environment. One of the goals of this architecture is to simplify the management of security by providing an abstraction of each device. This is done at the syndication tier by using a WAVE-enforcing information bus.

The design of this bus, WAVEMQ, is presented in Chapter 8. WAVEMQ is a tiered publish subscribe system designed to tolerate intermittent connectivity in addition to integrating WAVE. We evaluate WAVMEQ and discover that it has performance comparable to existing publish/subscribe systems whilst providing features such as denial of service resistance.

Chapter 9 discusses the implications of the security guarantees that WAVE provides and some of the associated concerns. Aside from complicating certain operations like auditing, WAVE also opens to door to new decentralized system design. Exploring this further, we consider the societal impact of the development of a privacy-preserving social media platform engineered around WAVE.

Chapter 10 extends the discussion of related work found in Chapter 2, and provides some examples of complementary work that could be incorporated into future WAVE versions. We also discuss work relating to the ULDM from Chapter 4 and WAVEMQ from Chapter 8.

We conclude in Chapter 11 with a summary of the dissertation, as well as a discussion of questions that arose throughout the research that may be of interest for future work.

Chapter 2

Background

Authorization is a problem that has existed since the first days of computing, so it has been the subject of study in both academia and industry for several decades. As a result, there is a body of work that can be broadly classified as providing solutions for authorization, although the specific design goals and choice of constraints vary significantly. This chapter gives an overview of some of these systems and how they differ in either desired functionality or security characteristics.

2.1 Requirements for a Decentralized Authorization System

Chapter 1 gives a high level view of the problems and scenarios that motivate this work. We now establish a concrete list of requirements and constraints, drawn from those scenarios.

High-Level Security Goal & Threat Model

One of the pitfalls of mainstream authorization systems, as identified in Chapter 1, is that the authorization system presents a central point of weakness. A compromise of the authorization system cascades throughout the rest of the system as the attacker is capable of masquerading as valid users with arbitrary permissions. Consequently, one of our high level objectives in this work is to design a system where the compromise of an authorization server does not compromise all the users and their permissions, leaving the integrity of other systems intact.

Namely, even if an adversary has compromised any authorization servers and users, it should not be able to:

1. Grant permissions on behalf of uncompromised users.
2. See permissions granted in the system, beyond those potentially relevant to the compromised users. See Chapter 5 and Appendix A for our definition of relevant.
3. Undetectably modify the permissions received/granted/revoked by uncompromised users from uncompromised users, or undetectably prevent uncompromised users from granting/receiving/revoking permissions to/from uncompromised users.

Failure Of Existing Systems

Existing authorization systems fall short in two general areas: they do not meet our security goals or they do not provide the features required for the usage scenarios in Chapter 1. Said simply, users need the ability to safely delegate permissions to other users, otherwise they are forced into insecure alternatives such as sharing accounts. More concretely, we summarize the following six requirements that are not simultaneously met by any existing system (as illustrated in Table 2.1):

No reliance on central trust. With systems relying on a central authorization server, compromise of that server allows for arbitrary credentials to be created. In essence all users are compromised. The requirement to avoid central trust is unmet by most existing systems. For example LDAP [34], which is widely used when authenticating people, and OAuth [91], which is common when authenticating and authorizing unattended services and devices, are both built upon a central server that stores authentication and authorization information without any cryptographic guarantees for integrity. There is no guarantee that the compromise of this server is even detectable, and the impact could spread beyond a single organizational boundary. For example, in the smart buildings scenario, the status quo has certain devices (e.g LiFx light bulbs) perform their authorization on the vendor's server in the cloud. If that server is compromised, all of those devices in all of the customer buildings are compromised. Similarly, the services using single sign-on, such as "Sign in with Google" will all be compromised if Google's service is compromised. In both of these cases, the adversary can violate all three security goals.

Transitive delegation. The smart building scenario illustrates the necessity for transitive delegation and revocation where, for example, a tenant can further delegate their permissions to a control service or guest and have those permissions predicated on the tenant's permissions. If the tenant moves out, all of the permissions they granted should be automatically revoked, even if the building manager is unaware of the grants the tenant has made. This form of transitive delegation is not found in widely-deployed systems like LDAP or OAuth: where delegation exists, it does not have this transitive predication property. This carries through to user-facing delegation in web services such as Dropbox or Google Docs where a user can delegate permissions to another user but the recipient becomes a first-class entry in the ACL with no record of how they obtained their permissions. Consequently, revocation of a user's permissions has no effect on any other users even if they received their permissions from the revoked user. In contrast, transitive delegation is well developed in academic work [95, 21, 84, 86, 55, 22, 99, 18].

Protected permissions. Parties should be able to see only the permissions that are potentially relevant to them. Even though the property manager is the authority for all the buildings, they must not be able to see the permissions that the tenants grant (Security Goal #2). Similarly one tenant should not be able to see the permissions granted to another. In contrast, because a device's permissions are predicated on all of the grants between the property manager and it, it should be able to see all of those relevant permissions. Existing systems do not offer a solution to this requirement: in many centralized systems, for example, whoever operates the server can see all the permissions.

Decentralized verification. Most existing decentralized systems, (e.g. SDSI/SPKI [95] and

Macaroons [20]) allow only the authority to verify that an action is authorized. This is adequate in the centralized service case where the authority is the service provider, but it does not work in the IoT case where the root authority (the property manager) has nothing to do with the devices needing to verify an action is authorized (for example a thermostat). Any participant must be able to verify that an action is authorized.

No ordering constraints. Delegations must be able to be instantiated in any chronological order. For example, a participant can delegate permissions in anticipation of being granted sufficient ones for the delegation to be useful. We have found this to be critical in our deployments as this pattern naturally occurs during installation of new devices. As a further example, when the building manager's key needed to be replaced (e.g. it expired or was compromised), they created a new key and the property manager had to grant replacement permissions to this new key. In many existing systems (e.g. Macaroons [20]), this necessitates every tenant re-creating their entire permission trees, as all grants must happen in sequence, following the grants to the replacement key. This is not tractable in practice as it requires the coordination of many people and hundreds of devices, leading to extended downtime. Furthermore, when we had such ordering constraints in our prior deployments we observed users choosing insecure long expiry times or broad permissions to avoid this re-issue. As a result, we require that the system enables permission grants to occur out of order, so that permissions grants can be modified (revoked / re-issued) or any key can be "replaced" without re-issuing subsequent delegations. We have also found that this capability leads to safer user practices as "mistakes" like overly narrow permissions and short expiry times are easy to correct.

Offline participants. Not all participants have a persistent online presence. A device may be offline at the time that it is granted permissions (e.g. during installation) and it must be able to discover that it received permissions when it comes online. This is trivial to solve with a centralized authorization system, but is not solved in existing decentralized systems (e.g SDSI/SPKI [95], Macaroons [20] and [21, 84, 86, 55, 53, 85, 117, 31, 110, 98]).

While many existing systems meet some of these requirements, no existing work meets all of the requirements concurrently.

2.2 Trust Management

The problem we are addressing is a subset of *Trust Management* (TM). Trust Management is an area of research that studies systems for negotiating trust between parties on the Internet through the exchange of credentials that encode both aspects of identity and aspects of authorization. Although individual credentials may have issuing authorities, the negotiation is typically conducted without an authority. Overviews of TM systems are provided in [22, 99, 18, 3].

TM literature acknowledges both the importance of transitive delegation, as well as cross-administrative-domain interactions, but the existing systems do not meet our goals either because the desired functionality is different or because the constraints are different. Table 2.1 summarizes related systems, including TM systems, and categorizes them into authorization languages, hidden credential systems, centralized authorization systems and distributed authorization systems.

Work	Transitive delegation	Discov-erability	No order constraints	Offline participants	No trusted central storage	Protected permissions
Auth. languages [20, 95, 21, 84, 86, 55, 53]	Yes	**No**	**Unknown: no mechanism given**			
Hidden credentials [113, 73, 58, 89]	Yes	**No**	**Unknown: no mechanism given**			
Centralized authorization [33, 23, 108, 54]	Yes	Yes	Yes	Yes	**No**	**No**
Distributed authorization [85, 117, 31, 110, 98]	Yes	Some	Yes	**No**	Yes	**No**
WAVE	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

Table 2.1: Related work on decentralized authorization compared to WAVE.

Authorization language work concerns itself with how complex authorization (and identity) policy is expressed [20, 21, 84, 14, 53]. While this is a piece of the problem and that work is complementary to WAVE, it does not aim to solve the problems we address. For example Macaroons [20] provides a mechanism for expressing authorization policy with delegation and context-specific third-party caveats, but the authorization is verifiable by the authority only and permissions can only be granted in-order. The system does not specify how cookies are stored and discovered or how it would work with offline participants.

In general, authorization language work is complementary to WAVE, as we focus on the layers of the system that lie below the language (how the pieces of policy are stored, disseminated, and discovered). In our deployments we use RTree, based on SPKI's pkpfs [45], as an authorization language, but mechanisms like third-party caveats could be introduced with no changes to the underlying layers and are a natural extension to this work.

Hidden credentials work [113, 73, 58, 89] addresses a different privacy problem: allowing a prover and verifier to hide their credentials and required policy from each other. This is valuable, and WAVE provides a simplified form of this functionality (see §5.10), but in general WAVE focuses on an orthogonal problem: the privacy of credentials in storage and during discovery. Credential dissemination and discovery is largely out of scope in hidden credentials work.

The remaining literature can be categorized as relying on a centralized credential store for discovery [33, 23, 108], or a distributed credential store [85, 117, 31, 110, 98].

Centralized discovery mechanisms put all credentials in *one place* which makes discovery simple but, as constructed in work thus far, requires this central storage to be trusted. Blockchain work [107, 50] avoids this problem by constructing the "centralized" store using distributed consensus, but does not scale and thus far has focused on identity, not authorization. Work, such as [54], decreases centralization by reducing the trust in cross-administrative-domain applications, such as IFTTT, but still places trust in the central authorization servers belonging to each vendor.

In contrast, distributed discovery mechanisms store each credential with its *issuer* and/or *subject*, avoiding the need to trust a central storage system. The resulting discovery mechanisms are more complex and cannot operate if credential holders are offline and cannot communicate credentials. Both the centralized and decentralized credential discovery work thus far have overlooked

the privacy of credentials in storage (in the centralized case) or during discovery (in the distributed case); in both cases, there are parties who can read credentials that do not grant them permissions even indirectly.

WAVE does not attempt to provide the functionality found in seminal authorization language and hidden credential work. Rather we are motivated to develop a middle ground that solves problems that TM work identified as important, while realizing a practical system that can replace widely used authorization systems, such as LDAP or OAuth.

2.3 SDSI & SPKI

To develop a decentralized authorization system, it is first necessary to identify a model for representing authorization that is compatible with the system goals. Most of the systems that we surveyed [20, 95, 45, 21, 84, 86, 55, 53, 113, 73, 58, 89, 33, 23, 108, 85, 117, 31, 110, 98] are incapable of operating without a central authority or require that participants are on-line.

SDSI [95], which solves problems of naming, and SPKI [45], which provides mechanisms for permission delegation were invented separately but joined forces to provide a comprehensive distributed authorization system. The system as implemented does not meet our goals, but it proposes a graph-based pattern of authorization that could potentially meet the requirements, with careful system engineering. We give a brief summary of the work here, as it is the first to introduce a comprehensive graph based authorization model.

In SDSI/SPKI, permissions begin at an ACL, which they also refer to as the list of root keys. This is similar to the ACL that is present in systems such as LDAP, but the entries in this ACL are public keys rather than username/password pairs. This allows a person (represented by a key) to delegate some of the permissions they possess by creating a signed certificate. The certificate identifies the recipient of the permissions by their public key, as well as which permissions are being delegated. This recipient in turn can delegate a portion of their permissions further by creating another signed certificate.

As an example, a photo sharing service would have an ACL with a list of all the users. A user A, present in the ACL, can delegate the ability to view some of their photographs to another user B by creating a certificate containing B's public key, a description of the permissions being granted and a signature made by A. When B wants to retrieve a photograph, it presents this certificate to the server and that server validates that A appears in the ACL and the certificate extending this permission to B is valid. B can also delegate to another user C in which case C would present both the $A \rightarrow B$ certificate and the $B \rightarrow C$ certificate.

In this sense, this is a prototypical graph-based decentralized authorization system as a given user can delegate a portion of their permissions with autonomy. That is to say the creation of the certificate did not involve the authority server holding the ACL. It is not a complete decentralized authorization system, however, as there are a few omissions from the design of the system:

1. There is a central ACL that contains the "first hop". Consequently, only the holder of the ACL (the service provider, or authority) is capable of verifying a proof of authorization (a

concatenation of a chain of certificates with an ACL entry). While the pattern of verification being done by the authority is common, it is not sufficient for use cases such as IoT where a device needs to verify that a command is authorized. In this case, the authority would likely be the building manager or company, not a single device within the building. This characteristic is not unique to SDSI/SPKI: Macaroons [20], a far more recent delegable authorization system, has the same problem. There is a secret used as the root of an HMAC chain that is held by the authority. As a result, only the authority can verify a given HMAC.

2. There is no mechanism for distributing or discovering certificates. This is out of scope in SDSI/SPKI but it is a challenging problem that has no obvious solution. Exchanging certificates in a peer-to-peer fashion requires participants to remain on-line in order to receive permissions. Storing certificates in a central repository introduces a central authority.

3. There is no mechanism for encrypting certificates in SDSI/SPKI. If storage is used as part of the solution for disseminating or discovering certificates, then there needs to be a mechanism for encrypting the certificates. This is considered out of scope for SDSI/SPKI but, as we discuss in Chapter 5, is a challenging problem.

We developed WAVE as a decentralized authorization framework that uses the graph based authorization model while solving these problems. It meets the above six requirements by introducing new storage and encryption techniques, as discussed in subsequent chapters. In doing so, we also answer the first thesis question:

> How can we construct an authorization system that supports first-class transitive delegation across administrative domains without trusting a central authority or compromising on privacy?

We will address the second thesis question, securing IoT systems, later in the dissertation (Chapter 7 and Chapter 8).

Chapter 3

Decentralized Graph Based Authorization

This chapter presents a model for capturing delegable authorization in a manner that is amenable to decentralization. While this has its roots in SDSI/SPKI as discussed in the previous chapter, we will present it using the terminology established by WAVE so as to maintain consistency as we discuss the mechanisms that permit its construction in subsequent chapters.

3.1 Concepts

Each of the components in Graph Based Authorization (GBA) are discussed in detail below. We begin with a high level overview.

GBA, as the name would suggest, represents authorization as a single global graph. Rather than more mainstream methods which have distinct data structures per domain of authority (such as ACL or role tables), GBA is logically a single graph that spans all domains of authority.

The vertices of this graph, called *entities*, represent anything that is capable of receiving or granting permissions. This could be something tangible, such as an individual person or device, but it can also represent digital participants, such as a computational service, and symbolic participants such as a group of people. Entities are identified by the hash of their public representation.

The edges of the authorization graph, called *attestations* represent delegations of permissions between entities. They contain identifiers for the granting and receiving entity, called the *issuer* and *subject*, as well as a description of the permissions being granted.

Permissions are granted on *resources*. Any given resource has, as part of its construction, an identifier for the *namespace* that the resource is a part of. The namespace is the authority entity that created the resource and therefore has full permissions on the resource without receiving them from any other entity.

An entity can form a *proof* of authorization for a given action on a resource by assembling a path of attestations that leads from the namespace entity for that resource to the entity forming the proof. The permissions granted by the proof are the intersection of the permissions granted by each attestation in the path through the graph.

3.2 Entities

An entity is a vertex in the global authorization graph that represents a participant in the system. Specifically, an entity is a collection of public keys that is referred to by its hash. Using techniques we will discuss later, an entity is a public object that can be looked up using its hash.

The secrets corresponding to an entity's public keys form the *entity secret* that is held by whichever party the entity represents. These key pairs allow for an entity to

- Create an attestation granting permissions to another entity
- Encrypt that attestation following the protocol described in Chapter 5
- Decrypt attestations that the entity can use to form a proof, also following Chapter 5
- Encrypt a message associated with a specific resource in a namespace
- Decrypt messages associated with resources the entity has permissions on

Concretely, an entity is an ASN.1 serialization of the following:

- An Ed25519 [17] key used for signing
- A Curve25519 [16] key used for encryption
- WKD-IBE [1] public parameters on curve BLS12-381 [26] used for Reverse Discoverable Encryption in Chapter 5
- IBE [24] public parameters on curve BLS12-381 used for Reverse Discoverable Encryption in Chapter 5
- A revocation commitment, as described in §3.5
- An expiry date
- A signature, using the Ed25519 key
- A random seed used for generating revocations

The IBE and WKD-IBE implementations on BLS12-381 are from [78]. An entity secret is the ASN.1 serialization of the corresponding secret keys, and is kept by the party that controls the entity.

3.3 Attestations

An attestation is primarily a policy that describes the permissions being granted. We begin by describing RTree, the policy mechanism used in WAVE. Additionally, to facilitate the privacy mechanisms in Chapter 5, the attestation must be partitioned into compartments that will later be encrypted. We present the attestation as it appears decrypted here and describe the encryption process in Chapter 5.

Resource Tree Auth Policy

Although WAVE is agnostic to the specific mechanism used for expressing authorization policy (i.e. allowing the use of existing policy languages such as [14, 20]), we use a simple yet common model throughout this work, for clarity. The resource tree policy mechanism, or *RTree*, is used for managing permissions on a hierarchically organized set of resources. It is a modification of SPKI's pkpfs tags [45].

A resource is denoted by a URI pattern such as `company-entity/project/folder/file` or `user-entity/albums/holidaypics/*`. The first element of an RTree URI (which would be `company-entity`) is called the *namespace* and it identifies the authority for the set of resources that has full permissions. The exact mapping from a resource URI to the artifact the resource represents is not proscribed by WAVE. For example, resources could be used to represent HTTP URIs or RPC methods when managing authorization for a service, but they can also be used to represent physical devices or symbolically to represent roles, as we discuss below. The $\star$ indicates a wildcard that can match anything, allowing a URI pattern to reference a resource subtree. Depending on the structure of a given resource hierarchy, there may be a minimum length for the resource URI. This often occurs where the first few elements are used to capture boundaries that exist naturally, such as a department, building or project. These elements that can be relied upon to exist, if present for a given RTree, are called the *resource prefix*.

Alongside the resource pattern, an RTree statement includes a set of permissions. The permissions are strings, such as "read" or "write", along with a unique *permission set* reference that identifies the permission schema. The introduction of the permission set is to prevent a permission granted by an entity in one context (for example "read" on archived sensor data) from being interpreted in a different context (a "read" command sent to a BACNET controller). Each application defines its own schema for what the permissions mean in their context and will ignore permissions granted in a different schema.

Finally, RTree has an *indirections* field, which limits delegation depth. An entity can grant access to another entity with zero indirections, preventing the recipient from further delegating the permission. This is similar to *propagate* in SPKI's pkpfs tags.

Thus, an RTree policy consists of:

- A set of permissions (strings such as "schema::read")
- A URI pattern describing a set of resources
- A time range describing when the grant is valid
- An *indirections* field, which limits re-delegation

Attestation Structure

An attestation contains three compartments. The first is the *plaintext compartment*, which is visible to all without performing any decryption. This contains

- The hash of the subject (recipient) entity
- The location identifier of the storage server the recipient entity is located on
- A revocation commitment (§3.5)
- An Ed25519 signature made by an ephemeral *outer* key (§5.4)

Then there is the *verifier compartment* which is visible to participants with the capacity to decrypt the attestation (both the entity forming the proof and the verifier who has received a proof and is checking that it is valid):

- The issuing (granting) entity hash
- The location identifier of the issuing entity
- The expiry of the attestation
- The policy, e.g. RTree statements
- A signature made with the issuer's Ed25519 key of the outer ephemeral key

Finally there is the *prover* compartment, visible only to the entity forming the proof, which contains cryptographic material specifically for implementing Reverse Discoverable Encryption (Chapter 5) which we will discuss in that section.

3.4 Underlying System Requirements

To meet the requirements established in §2.1, we need two components of the system that will be developed in subsequent chapters.

The first is a global storage system, called an Unequivocable Log Derived Map (ULDM), that allows us to place arbitrary objects in storage and identify/retrieve them by hash. WAVE provides this in a manner that, by construction, does not introduce a central authority. This is developed in Chapter 4.

The second is Reverse Discoverable Encryption (RDE), which ensures that the attestations placed in the above storage are only visible to entities that are capable of using those attestations in a proof. This means that despite the storage being public with no access control, we can assume that attestations that do not concern a given entity either directly or indirectly, remain invisible to that entity. This technique is developed in Chapter 5.

3.5 Revocation

When a user creates an attestation, it derives a random revocation secret s from a seed stored with the entity secret and includes a cryptographic hash of s, $\mathsf{hash}(s)$, called the *revocation commitment*, in the attestation. The attestation is then published in the ULDM storage tier. Later on, the user can revoke the attestation by publishing the revocation secret s to the same ULDM storage tier. Revocation of entities works similarly. An entity must have their private key to perform revocation; mechanisms such as [100] can be used to ensure this.

When verifying a proof, the WAVE service ensures that no attestations in the proof have been revoked. To do so, it queries the storage tier for an object matching the revocation commitment $\mathsf{hash}(s)$ in the attestation. If such an object exists, the verifier knows that the attestation has been revoked. If such an object does not exist, the verifier receives a *proof of nonexistence* for that hash from the storage tier. As the storage permits retrieving object by hash, checking if the revocation commitment exists in storage is equivalent to checking if the attestation has been revoked. The integrity guarantees of the storage tier, described in Chapter 4 carry through to the revocation scheme.

Alternatively, the entity forming the WAVE proof can include the assertion returned by the ULDM storage tier that states that the revocation does not exist. Including this with the attestations in the WAVE proof means that the verifier does not have to look up the revocation itself.

3.6 Proofs

A WAVE proof is a self-standing object that can be verified to yield an authorization policy. Concretely, it is a list of attestation edges in the global authorization graph leading from the authority entity for the resources in question to the proving entity. Each policy mechanism has a way of identifying the authority entity. For RTree it is the namespace, so an RTree proof is a list of attestations assembled end-to-end forming a path from the namespace authority to the prover. The authorization policy yielded by the proof is the intersection of the policy granted in each attestation. The indirections field is checked (much like a TTL in IP networking) to ensure the maximum indirections of any given attestation is not exceeded. The proof does not require any additional resolution to verify and it cannot be tampered with or modified (as each attestation is signed by its issuer). As we examine privacy concerns, the proof will be extended to contain the necessary keys to decrypt its constituent attestations.

In systems using WAVE, a proof is typically attached to each message so that the actions within that message can be compared against the proof policy and the message recipients can determine if the message is valid and should be acted upon, or if the message is invalid. To verify that the proof concerns the message creator, the message is typically signed by the same entity that appears in the proof as subject.

For performance reasons, a proof may also be used to establish the authorization policy for a session and subsequent messages identify themselves as part of that session either by being part of the same TCP flow or by using a simple mechanism, like a shared secret.

Advanced Proof Patterns

Thus far, we have discussed the proof as being a single path through the authorization graph. While many proofs are in fact of this form, it is sometimes necessary to construct a proof that is a subgraph, rather than a linear path. As an example, consider a case where an entity A grants an entity B multiple different permissions contained within different attestations over a span of time, and B grants all of these permissions to an entity C in a single attestation. It would be convenient for C to be able to form a single proof that proves multiple permissions, yet there is no singular path that proves these permissions. Rather, for the link from A to B, C needs to be able to include multiple attestations in parallel.

The combined subgraph proof is faster to build, more concisely representable and faster to verify than a concatenation of the individual paths that would prove the same permissions. We discuss this more in §6.9.

3.7 Common Authorization Modes

While RTree is formulated around managing permissions on resources, creative construction of the resource trees allows for RTree to be used for a variety of authorization tasks. This section gives patterns for using graph based authorization and RTree to achieve common authorization modes.

The two most common types of authorization are Discretionary Access Control (DAC) and Role Based Access Control (RBAC). Both of these can be achieved using a graph based authorization pattern.

Many subtly-different authorization patterns fall under the banner of DAC. In general, if the pattern is built around a table that lists users or groups with the permissions that they have, or a table that lists resources with the users or groups that can access them, then it is likely classified as DAC.

In contrast, RBAC does not maintain permissions for a given user but rather maintains a list of roles that a user may assume (or a policy that dictates under which conditions a user may assume a role). Depending on the system, access control policy may be expressed directly as roles (there is no notion of resources, only that a certain role may interact with the service the resources are associated with) or there may be a distinct table mapping roles onto the resources the role may interact with.

Discretionary Access Control

Examples of DAC include POSIX file permissions, which states which groups have which permissions on each file, and most online document and file sharing suites such as Google Docs, Dropbox, etc. GBA and the RTree policy type are very well suited for implementing DAC.

In a resource-oriented DAC system where the table records with users have which permissions on which resources, the operations that a DAC system must offer are:

1. Grant a user permissions on a resource

2. Remove a user's permissions on a resource

3. Determine if a user has permissions on a resource

In DAC, there is not always a clear analogue to the GBA namespace authority. There is usually a user that has permissions on a resource without obtaining them from another user (root in POSIX, the creator of the document in Google Docs) but there is sometimes more than one user that can give permissions to another user without having them themselves. For example if the owner of a POSIX file does not have any permissions on a file, that does not prevent them from changing the permissions and granting themselves (or other users) access to the file that they did not have. In systems like Google Docs or Dropbox this is not a meaningful distinction because the ability to edit the DAC table is bundled with the ability to read and write the document.

Let us consider a file server as an example. Imagine there is a directory `/home/bob/photos` that is owned by Bob and he wishes to grant read access to Alice. We have two ways of formulating the resource hierarchy when we map this to WAVE. The first places the server operator as the namespace authority (i.e. the URI would be `server-entity/bob/photos`). This is closer to the traditional authorization pattern used by most file servers, and would work, but it is not ideal as it makes the server operator an authority over all the users. Alternatively, we can allow each user to be their own namespace, which removes the server operator from a given user's authorization graph. This is preferred as we have now removed a central authority. In this case, the WAVE URI would be `bob-entity/photos` and the server operator does not have the ability to grant permissions on the resource. Only Bob, and those to which Bob has delegated this right to, are capable of granting permissions on the resource.

The DAC functionality then maps onto WAVE operations as:

1. Granting a user permission on a resource is done by creating an attestation. If the ability to re-delegate is desired, the attestation is created with a nonzero indirections field.

2. Removing a user's permissions is done by revoking a created attestation.

3. Users interacting with a resource on the server must attach a WAVE proof to each request and determining if the user has legitimate permissions is done by validating the WAVE proof.

Role Based Access Control

It is more complex to implement RBAC using WAVE and the exact pattern depends on the manner in which the RBAC itself is formulated. We consider two different types of RBAC here and give a mapping for each.

Role-Oriented RBAC

The first RBAC type treats roles as a first-class, meaning that it does not perform a subsequent mapping from roles onto permissions and resources, but rather expresses all authorization policy

in terms of roles. In this formulation, a service has a local configuration that lists which roles are permitted to perform which actions and the external authorization system only manages which users are permitted to assume which roles.

The functionality required by the RBAC system is therefore:

1. Record that a user is allowed to assume a role
2. Remove the ability for a user to assume a role
3. Allow a user to assume a role and prove they are allowed to do so

Traditionally (3) above is done by having the service provider contact the authorization server to verify the user can assume the given role, but there are also bearer-token systems that allow the user to prove it has assumed a role without requiring communication with the authorization server, such as RBAC using OAuth [91].

We can map this onto WAVE as follows: the resource tree becomes a hierarchy of roles. For example `namespace/role/auditor/logs` for a log-auditor role and `namespace/role/auditor/*` for an anything-auditor role that is a strict superset of the log-auditor role. The operations then become:

1. To allow a user to assume a role, the namespace entity grants an attestation with the permission `rbac::assume-role` and a resource pattern that indicates the roles that can be assumed.
2. To remove a user's ability to assume a role, the attestation is revoked.
3. To prove it can assume a role, the entity forms a GBA proof that it has the `rbac:assume-role` permission on the role it is trying to assume.

This formulation has the advantage that if a user is allowed to assume a role but also allowed to grant other users the ability to assume that role, that can be captured by creating the attestation with a nonzero indirections field.

Permission-Oriented RBAC

There is another pattern of RBAC where roles are just an intermediary and the role maps onto a set of permissions on resources as in DAC. This is, for example, the pattern used in Amazon Web Services Identity and Access Management RBAC. It is not clear if this particular pattern should be preserved when moving towards delegable authorization, but it is possible to do so.

In this formulation, a role is represented by an entity (R1 and R2 in Fig. 3.1), and permissions and resources are represented in RTree policies as with DAC. To assign permissions to a role, one grants an attestation to the role entity. To allow another entity to assume a role, one creates an attestation from the role entity to the assuming entity with a wildcard permission and resource, as shown in Fig. 3.1. This also allows a role to assume another role. Essentially all permissions granted to the role entity will transfer to the assuming entity. The assuming entity does not form

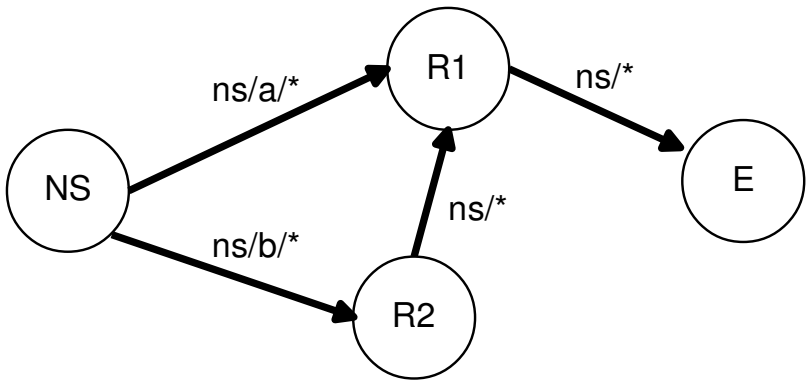


Figure 3.1: Permission-Oriented RBAC: Two roles R1, R2 where R1 is a superset of R2. E has been granted the ability to assume R1 and, transitively, R2. The namespace has granted permissions to the roles directly, but they can also be granted transitively

a proof that it can assume the role, but rather skips the intermediate step and forms a proof of the desired permissions, which will be a path that flows through the role it is assuming to obtain those permissions.

While this may not be RBAC in the strictest sense, it preserves the ability to do transitive delegation and allows entities to act with autonomy, which a centralized RBAC system does not.

Identity Claims

A common task in Trust Management literature is providing identity claims to a peer in order to establish trust, such as proving that you are an employee of a company. While a focus of WAVE is proving authorization without bootstrapping trust through authentication, it is possible to perform rudimentary identity management using GBA and RTree.

To do so, we model identity factors (such as membership of an organization or possession of a license) as resources in RTree, and what would typically be the certificate authority as the namespace.

For example, to prove membership of an organization where existing members are permitted to recruit new ones, one might grant `org::ismember` on `organization/chapter`. A new member or employee might not have the authority to add new members, so the attestation would be created with no indirections. A more established member might be given the ability to recruit members, and therefore their grant would have nonzero indirections.

This pattern can be used for arbitrary identity claims such as `dmv/licensedDriver` or `dmv/over21` but the notion of delegation is often not meaningful in these cases as, for example, being able to prove you are a licensed driver does not usually mean the DMV trusts you to certify that another person is a licensed driver.

We mention these because by combining these identity claims with the anonymous proof mechanism in §5.10, it is possible to prove aspects of your identity without revealing who you are. This is a common goal in TM literature.

3.8 Graph Based Authorization Evaluation

WAVE's performance characteristics are largely determined by the cost of the mechanisms associated with preserving privacy. We characterize the performance of these mechanisms in Chapter 5 and the performance of the implementation as a whole in §6.10.

This section focuses on evaluating the efficacy of graph based authorization as an authorization pattern, and the intrinsic performance characteristics that the pattern implies, rather than on side-effects of implementation choices.

Advantages of Graph Based Authorization

We discussed how GBA is capable of implementing patterns such as Discretionary Access Control and Role Based Access Control, but this does not fully convey why one would choose GBA over other, simpler, mechanisms for realizing DAC or RBAC.

The primary reason that we chose GBA is, of course, that it is possible to realize a decentralized GBA system with desirable security characteristics whereas we were not able to achieve those characteristics with other patterns. This is not the only benefit, however. When using GBA in practice we discovered that it improved workflow and is a desirable pattern even if one did not care about the improvements to security.

Traditional systems protected by DAC have a fairly monolithic architecture: there is a cluster of servers administered by one logical entity, so a single authorization list is not incongruous. The same is not true of IoT or the built environment as a whole. There are far more parties involved in a typical IoT scenario, and this places strain on the central pattern, leading to fragmentation of authority. Consider that many IoT devices are associated with a service: a smart doorbell requires the vendor's cloud and would not function without it. Similarly a digital assistant like Google Home or Amazon Alexa is intricately tied to a continuously evolving backend hosted by Google or Amazon. Ownership of such a device is a murky concept: for better or for worse, the device does not completely fall under the authority of the customer when they purchase it. As a consequence, a typical smart building is an assemblage of devices interwoven with vendor and customer services, each with their own authorization mechanisms. This in turn has led to the development of services that attempt to abstract away some of this complexity, such as AllJoyn [5]. Unfortunately, AllJoyn makes use of a centralized "Security Manager", which does not meet our requirements.

The decentralized management of such a collection of devices is extremely challenging. In Chapter 7 we discuss how an abstraction layer can permit us to unify these devices and services. Given that this affords us the opportunity to present a different authorization interface, we can consider alternate patterns and their merits. Taming this heterogeneity would typically require the construction of a single unifying authorization policy hosted on a central system within a single domain of authority, which has thus far proven difficult to achieve. Graph based authorization allows for the problem to be broken down into smaller, more manageable pieces. Authority is delegated to parties that can then act with autonomy, determining their own authorization policies that best fit within their context. This gives us flexibility to deal with the heterogeneous landscape of IoT devices and control services. As a concrete example, when a new set of sensors and their

associated control services is installed within a building, the team heading that install is granted the authority to associate the sensing and control points with a subtree of the resource hierarchy in the building. The installing team is free to determine which devices and services have which access within this subtree, and they can do so with full autonomy—the authority for the building is not involved in this process. After the installation process is done, the authorization subgraph built by the installing team is re-rooted into a permanent position within the building's authorization graph by revoking access to the installing team's entities and granting direct access to the persistent entities representing the collection of devices and services that have been installed.

The introduction of autonomy is a key benefit of graph based authorization. In practice it obviates coordination between multiple parties that would typically slow down any changes to authorization. Consequently, it allows users to develop very specific, narrow authorization policies that evolve over time as the needs of a service or device change. Historically, one would obtain a very permissive grant for a service or device in an attempt to avoid going through the painful process of changing permissions, which results in decreased security. Overly permissive policies increase the severity of a compromise as a device or service is able to affect systems outside of its immediate context. This aversion to changing policy is especially true if the authorization crosses administrative domains, where changes may manifest as changes to business contracts. The Target hack [77] was a good example of this where the HVAC company did not expend the effort to get Target to configure narrow network policies for their systems but rather obtained broad access, as it was easier and required less negotiation between the companies. Consequently, compromise of the HVAC system cascaded throughout Target's systems and resulted in large scale data breach that could have been avoided had appropriately narrow policies been put in place.

Performance of Graph Based Authorization

The key operations in an authorization system are modifying a party's permissions and determining the permissions that a party has. In graph based authorization, both of those are split into two, so there are four operations:

1. Granting permissions
2. Discovering permissions
3. Forming a proof of authorization
4. Verifying a proof of authorization

We will consider the intrinsic costs of each of these operations here, but we additionally provide real characterizations of these operations, as done by a full implementation, in subsequent chapters.

Granting permissions in GBA requires the construction of an attestation that represents the policy being delegated. Once the attestation has been constructed, it must be published into global storage. A traditional DAC system would require a sequence of network interactions with the authorization server: first to establish that the granting party has access to the server and then to modify the

permissions associated with the recipient party. GBA requires a single interaction with the global storage to place the attestation in storage. There is not an initial authentication phase in GBA as the global storage permits any party to store attestations. Verification is done when the attestation is discovered, i.e. during proof construction. As a result, we expect the granting of permissions in GBA to be of similar performance to that in a traditional authorization system. In WAVE, the privacy mechanism introduced in Chapter 5 adds cryptographic operations to the formation of attestations. This makes permission granting slower than in traditional systems, but still below the latency that would be significant to a user.

Discovering permissions in GBA requires traversing the global graph to find the attestations that concern you. This is intrinsically expensive compared to discovering permissions in a DAC system. Traversing the global graph captures a sequence of granting events that must be processed to determine the final permissions, and one does not know in advance if the path being traversed will eventually end in the proving entity (making it relevant) or not. In contrast, an ACL has already processed all granting events and only stores the results, indexed by principal. In WAVE, the performance of graph traversal is further complicated by the global graph being encrypted, as each attestation must be decrypted after it is discovered. The encryption has the advantage that less of the graph is visible, so one does not need to traverse the whole graph looking for attestations that are relevant. This privacy comes with a cost, however, as the decryption operations themselves are computationally intensive. In WAVE, permission discovery is the most expensive operation, but remains reasonable for normal use cases.

Forming a proof of authorization in GBA requires the discovery of a path through the decrypted subgraph of relevant attestations assembled during permission discovery. This is a straightforward shortest path discovery problem, so the run time will depend on the length of the path and the topology of the decrypted subgraph. A traditional system often places this burden on the service rather than the client (i.e. the client sends a username and password and the service performs the lookup) but we can compare to systems such as OAuth where the client first obtains a proof of authorization (a token). Given that OAuth requires a network round trip and GBA does not, it is possible for GBA to be faster than OAuth. In addition, proofs can be cached until a constituent attestation expires in GBA whereas tokens in OAuth typically have shorter lifetimes. Therefore we expect forming a proof of authorization in GBA to be comparable or faster than similar operations in existing systems.

Verifying a proof of authorization in GBA requires checking that the policies in the attestations are compatible with each other. This can be done with no network round trips (depending on how strict the revocation policy is). We expect that verification of a GBA proof would be substantially faster than the authentication and subsequent authorization operations found in a system such as LDAP which requires multiple network round trips. Verification of authorization scopes in OAuth requires checking that a token is valid, which is a similar operation to checking a GBA proof, so we expect those to have comparable performance. In general we expect that verifying a proof of authorization in GBA will be fast enough to be imperceptible for all applications migrating from traditional methods.

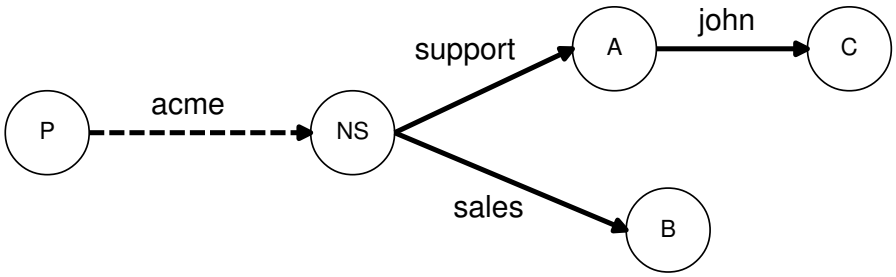


Figure 3.2: An example name declaration subgraph. The dotted line indicates a local name. The solid lines indicate global names.

3.9 Name Resolution

To facilitate looking up entity public keys (to be used as the subject in an attestation, and for RDE), without relying on an external PKI, WAVE implements a naming scheme that extends the proposal in SDSI [95]. The base functionality (shared by WAVE and SDSI) allows an entity to name another entity by creating a signed *name declaration*. These name declarations form a web-of-trust global graph, similar to that formed by attestations.

Consider Fig. 3.2. Here, a namespace entity `NS` has created a name declaration that calls `A` "support". `A` has independently created a name declaration naming `C` "john". In addition `NS` has named `B` "sales". From the perspective of an entity `P`, who has named `NS` "acme", one can traverse this graph to resolve a name into an entity for example acme $\rightarrow$ support $\rightarrow$ john. To increase similarity with DNS names that users are already familiar with, we reverse the names, placing the first name declaration on the right, and we separate each name with a ".". The above name would then be john.support.acme.

The name declaration made by `P` that names `NS` "acme" need not be published and can be kept local to just `P`. The other name declarations must be published by the entities creating them, if the creators wish them to be resolvable by other parties.

The advantage of this system is that by verifying the identity of a single entity out of band (e.g. the company namespace entity), an entity can resolve the names of all employees within the company's departments, such as john.support.acme, without having to manually establish the validity of individual employee entities.

An additional advantage is that these names also express a flow of trust in identity. If `P` trusts `NS` they are also implicitly trusting that the names that `NS` creates are also valid (that `NS` would not name an attacker entity "marketing" for example). We achieve trustworthy names without requiring a central authority to maintain names, unlike TLS Certificate Authorities and DNS. The naming system has the same autonomy characteristics as the delegable authorization described above: an entity can name a person, device or service without contacting an authority hosting a central directory.

The concepts above, which are very similar to the functionality proposed by SDSI, do not provide a distribution mechanism for entities to discover the name declarations required to perform

resolution, nor a mechanism to ensure the privacy of declarations so that only authorized parties may read them. WAVE solves both of these problems. Firstly, WAVE stores name declarations in the ULDM storage tier (Chapter 4) to ensure name declarations are discoverable without compromising on the goals of the system (especially without requiring on-line participants). Secondly, WAVE uses the encryption scheme described in §5.11 to encrypt the name declarations in storage. When creating a name declaration, it is associated with a resource in a namespace (for example, `acme/directory/marketing`) and an entity must be explicitly granted permission on that resource in order to gain the keys required to decrypt the name declaration. In other words, the same attestations that are used to form a proof of authorization are also used to govern which entities can read name declarations, without relying on a central directory server. Resolution of names is done from each entity's cache of decrypted name declarations, stored alongside decrypted attestations.

3.10 Summary

Graph based authorization is a versatile technique built around a single global graph that represents participants as *entities* which are vertices in the graph and the granting of permissions as *attestations* which are edges in the graph. A proof of authorization is a path through this graph. This proof is typically attached with every action or message to demonstrate it is authorized. GBA is capable of subsuming popular authorization patterns such as Discretionary Access Control and Role Based Access control, and it is possible to build a GBA system with better security characteristics than traditional DAC or RBAC systems. GBA allows for entities to act with autonomy, which simplifies management of heterogeneous devices and services, especially across administrative domains. Most operations in GBA are expected to have comparable or better performance characteristics than their traditional counterparts with the exception of permission discovery which, while slower, is expected to remain reasonable.

Realization of a GBA system requires the development of a secure global storage system where attestations and entities can be kept, enabling permission discovery. In addition, GBA requires an encryption mechanism to restrict the visibility of attestations in storage to just those parties that can use them in a proof. These two components are developed in Chapter 4 and Chapter 5 respectively.

Chapter 4

Secure Storage

WAVE allows for permissions to be granted to an entity without communicating with that entity, which allows the receiving entity to be offline at the time of the grant. This functionality leads to the requirement for a means by which WAVE objects, such as attestations, can be stored and retrieved. This chapter describes the storage tier that fulfills this role.

Certain high level security goals of WAVE are achieved through the storage layer, and inherited by other components. Hence, the requirements placed on the storage layer are more complex than simply storing and retrieving would suggest. We outline these requirements in §4.1 and give a standalone rationale for each. Some of them will become more clear when considered in the context of mechanisms presented in subsequent chapters.

The storage requirements and corresponding solutions have evolved over the course of WAVE's history, but there are lessons to be learned from the earlier solutions, so we address all three stages in the evolution of the storage subsystem.

4.1 Operational and Security Requirements

The storage system must support a set of operations required by WAVE, and it must be able to produce four types of proofs that these operations have been carried out honestly.

Operations

The graph based authorization model relies upon the notion of objects that represent nodes (entities) and edges (attestations) in the global graph. To permit interaction with the graph, the storage layer must allow participants to perform the following six operations.

1) Store/Retrieve Entities. Recall that entities have a public and private component. The public component of an entity is a collection of public keys and an expiry. This collection is referenced by the hash of the serialization of the keys and expiry. For many operations in WAVE, such as granting an attestation or verifying a proof, an entity will need to retrieve the public entity objects given only the hash. This public object is uploaded by the creator of the entity. From the perspective of the

storage, the entity is an opaque blob. The storage does not index any parts of the entity other than the hash of the object. Concretely, this requires the following actions on an abstract storage state:

$$\texttt{Put}(Entity) : state \leftarrow state \cup Entity$$
$$\texttt{Get}(Hash(Entity)) : Entity$$

2) Store/Retrieve Attestations. The succinct serialization of a proof only includes the hashes of the constituent attestations. When verifying such a proof, a participant needs to be able to fetch the attestations given their hash. After creating an attestation, the issuer uploads the encrypted attestation to storage. Like entities, the attestations are treated as opaque blobs by the storage, only accessed by hash:

$$\texttt{Put}(Attestation) : state \leftarrow state \cup Attestation$$
$$\texttt{Get}(Hash(Attestation)) : Attestation$$

3) Store/Retrieve Name Declarations. Much like attestations, participants need to retrieve and decrypt name declarations in order to maintain their local index of human readable names to entities (§3.9). These are also only accessed by hash (not by the name they are binding).

$$\texttt{Put}(NameDecl) : state \leftarrow state \cup NameDecl$$
$$\texttt{Get}(Hash(NameDecl)) : NameDecl$$

4) Notify Parties of Attestation. When creating an attestation, the issuer must associate the newly created attestation with a list of attestations granted to the subject, so that it can be discovered by entities that have keys from that subject. As the storage treats the attestation as an opaque blob, the subject is provided by the clients with the request. The set of entities that are interested in the attestation includes the subject of the attestation, but also entities that have been delegated permissions (possibly indirectly) by the subject entity that intersect with the permissions of the newly created attestation. Note that neither the issuing entity nor the storage knows the set of entities that will find the new attestation useful. The mechanism relies on the fact that entities that have discovered keys from the attestation subject (and therefore will find attestations granted to that subject useful) will periodically check the storage for new attestations associated with the subject.

$$\texttt{Notify}(attestation) : nstate_{attestation.subject} \leftarrow nstate_{attestation.subject} \cup attestation$$
$$\texttt{GetNotifications}(attestation.subject) : nstate_{attestation.subject}$$

5) Notify Parties of a Name Declaration. As with attestations, a newly created name declaration must be discovered by the set of entities that could resolve the name. Unlike attestations, however, the entity that is interested in the name declaration is the issuer of the name declaration, not

the subject. Briefly, this is because hierarchical names are discovered by traversing the graph *forwards* from namespace to named entity, whereas attestation paths are traversed backwards during discovery. We discuss this more in §3.9. This requires the following operations:

$$\texttt{Notify}(NameDecl) : nstate_{NameDecl.issuer} \leftarrow nstate_{NameDecl.issuer} \cup NameDecl$$
$$\texttt{GetNotifications}(NameDecl.issuer) : nstate_{NameDecl.issuer}$$

6) Store/Retrieve Revocations. The revocation scheme described in §3.5 works by having a revocation object with a known hash in storage. If the object exists, then the entity or attestation containing the corresponding revocation object hash is considered revoked. If the object does not exist, then the corresponding entity or attestation is still valid. This requires the following operations:

$$\texttt{Put}(Revocation) : state \leftarrow state \cup Revocation$$
$$\texttt{Get}(Hash(Revocation)) : Revocation$$

Proofs of honest operation

In addition to supporting the above operations, the storage tier must also be able to provide cryptographically verifiable proofs that each operation has been performed honestly. This is required so that the logically centralized storage tier does not become a central point of weakness (i.e. a component that must be trusted in order for the whole system to remain trustworthy). We cannot construct a proof that the storage provider is maintaining the privacy of the data, so we solve that problem using the encryption mechanisms in Chapter 5. The following classes of proof let the storage provider show that it is maintaining the integrity of the storage state:

1) Proof of Monotonicity. The storage must be able to prove that it is only adding objects to the storage, never removing them or modifying existing objects. This is especially important for revocations as an attacker would benefit from removing the revocation object for a compromised key. This proof is necessary for the `Put` and `Notify` operations. Both of these operations yield a state change $state' \leftarrow state \cup (object)$. A proof of monotonicity shows that only the given object has been added to the state and no other change been applied to yield $state'$. A proof of monotonicity can also be formed between two states that are multiple changes apart, and this is equivalent to a sequence of proofs of monotonicity showing that each individual change was monotonic.

2) Proof of Inclusion. When an object is added to storage or retrieved from storage, the storage must prove that the object really exists in the storage. A similar requirement exists for notifications: when a notification is inserted or retrieved from storage, the storage must be able to prove that the notification event exists. This is required for all six operations. A proof of inclusion shows that $state \supset object$.

3) Proof of Non-Existence. When a user asks for an object that has not been stored, the storage returns $\perp$. A proof of non-existance shows that $object \not\subset state$ and hence that the $\perp$ return value

is correct. The proof of inclusion and proof of non-existence can usually be done with the same mechanism.

4) Proof of Non-Equivocation. Equivocation is when a server presents different internally consistent views to different users. The storage must be able to give some proof that every user is seeing the same data structure. The nature of this proof varies depending on how the storage tier is implemented.

These proofs ensure that the storage provider is maintaining the integrity of the global authorization graph. In turn this allows us to treat the, potentially centralized, system as not being a central authority.

4.2 Abstract Implementation of Operational Requirements

An abstract implementation of the six required operations can be realized with a key/value store that permits chosen keys. This can be used to meet the functional requirements as follows:

Storage/Retrieval of Objects

Operations 1,2,3 and 6, rely on the ability to store on object and retrieve it by hash. This is quite simple to achieve with a key/value store. The key is the hash of the object, and the value is the object itself. No special handling of the different operations is required as the storage does not distinguish between entities, attestations and name declarations. We need a special $\perp$ exception that can be returned when a client attempts to retrieve an object that does not exist. There are numerous systems that offer a key/value store that is suitable, so the choice of implementation will be driven by the requirements to form proofs. Given a key/value store, the operations can be implemented as:

function PUT(object)
    $state[\ Hash(object)\ ] \leftarrow object$
end function
function GET(Hash(object))
    if $\exists state[\ Hash(object)\ ]$ **then**
        return $state[\ Hash(object)\ ]$
    else
        return $\perp$
    end if
end function

Note that the storage layer has no notion of access control; any participant can retrieve any entity, name declaration, revocation or attestation from storage (if it knows the hash). The storage is enumerable, so we also do not gain security from having a sparse key space where keys are hard to guess. We rely on the privacy mechanism in Chapter 5 to ensure that entities cannot read the objects they are not supposed to.

Notification

Operations 4 and 5 require maintaining a set of notification events. Notification does not map onto a key/value store as simply, but one approach is to maintain a list of notification events for each entity. The list is constructed as a set of objects with their keys chosen to allow iteration. For attestations, notifications are associated with the subject of the attestation. For name declarations, notifications are associated with the issuer of the name declaration. The associated entity hash is used as part of the key prefix and a monotonically increasing integer as the key suffix:

function NOTIFYATTESTATION(attestation)
  $sub \leftarrow attestation.subjectHash$
  $prefix \leftarrow$"notificationAttestation"
  $id \leftarrow 0$
  while $\exists\ state[\ prefix \| sub \| id\ ]$ **do**
    $id \leftarrow id + 1$
  end while
  $state[\ prefix \| sub \| i\ ] \leftarrow Hash(attestation)$
end function
function GETNOTIFICATIONATTESTATION(subjectHash, id)
  $sub \leftarrow subjectHash$
  $prefix \leftarrow$"notificationAttestation"
  if $\exists state[\ prefix \| sub \| id\ ]$ **then**
    return $state[\ prefix \| sub \| id\ ]$
  else
    return $\perp$
  end if
end function
function NOTIFYNAMEDECL(namedecl)
  $iss \leftarrow namedecl.issuerHash$
  $prefix \leftarrow$"notificationNameDecl"
  $i \leftarrow 0$
  while $\exists\ state[\ prefix \| iss \| i\ ]$ **do**
    $i \leftarrow i + 1$
  end while
  $state[\ prefix \| iss \| i\ ] \leftarrow Hash(namedecl)$
end function
function GETNOTIFICATIONNAMEDECL(issuerHash, id)
  $iss \leftarrow issuerHash$
  $prefix \leftarrow$"notificationNameDecl"
  if $\exists state[\ prefix \| iss \| id\ ]$ **then**
    return $state[\ prefix \| iss \| id\ ]$
  else
    return $\perp$

end if
end function

To obtain the entire set of notification events, the client will iterate through id values until it obtains $\perp$. To update the set of notification events, the client can resume iteration beginning with the id that mapped to $\perp$ on the last update. Portions of this procedure can appear on the client side, for example, the storage provider can expose just put/get of arbitrary keys and the client can perform the iteration.

Although implementation dependant, the proof requirements usually lead to the storage provider linearizing operations applied on the storage state. If exposing just a put/get API, however, it is possible that after a client A identifies an empty slot through iteration, another client B inserts into that slot before a client A's final insertion request hits the storage server. In this case, the storage server responds with an error and an inclusion proof showing that the (previously empty) slot that A attempted to assign to is occupied and A will try the next slot. As objects are inserted into the storage infrequently, this simple retry mechanism is sufficient.

When an entity A receives permissions from another entity B, A will add B's notification queue to the set of queues that A needs to check when looking for new attestations. This allows A to discover attestations that are not directly issued to them, but are useful because they grant permissions to an entity like B that has already granted permissions to A.

For name declarations, if X names entity Y who names entity Z, then an entity A that chooses to resolve hierarchical names under the namespace X will add X's notification queue to the set of queues that need to be checked, and iteratively add the queues for Y and Z as it discovers the name declarations that point to those entities.

Implementing these protocols whilst dealing with concurrency and being able to provide the integrity proofs is challenging. This led us to build on top of a system that already solves those problems: blockchain.

4.3 Blockchain Storage

WAVE version 2 provides the first complete solution to these requirements in the form of blockchain smart contracts. Ethereum's smart contracts can execute complex logic and store state in a cryptographically verifiable manner that allows us to very succinctly implement the abstract key/value based scheme listed above. From a security perspective, Ethereum is physically and logically decentralized: it is spread over multiple servers owned by different parties. This lets Ethereum meet the decentralization goal in §2.1, despite presenting a simple interface that appears like the contract is being run on a single controlling node.

Minimal Contract Solution

While blockchains are typically associated with facilitating financial transactions, and smart contracts are associated with autonomously managing tokens or funds of some sort, we use the

blockchain in a different way. The contracts supporting WAVE store the global graph by using the blockchain as a decentralized key/value store.

The clients that participate in the blockchain can interact with the smart contracts deployed on it, and therefore can use the blockchain client to access the global authorization graph.

The following Ethereum Solidity contract implements the abstract key/value store scheme:

```
1  contract Registry {
2   mapping(bytes32 => bytes) public objects;
3   mapping(bytes32 => bytes32[]) public lists;
4
5   function put(bytes data) public{
6     objects[keccak256(data)] = data;
7   }
8   function append(bytes32 lst, bytes32 h) public{
9     lists[lst].push(h);
10  }
11 }
```

In the above code, the `objects` and `lists` mapping declarations on line 2 and 3 declare persistent global variables in the smart contract. Anything that is stored in these variables will be stored in the blockchain state forever. Solidity will generate accessor functions that allow clients and other contracts to read these variables.

In addition to meeting the functionality requirements, the use of Ethereum also makes this approach fulfill the security requirements. In the typical configuration, every client interacting with the blockchain maintains a full copy of the blockchain contents. When a new transaction is applied to the blockchain, every client receives that transaction and applies it to their local copy. The proof of monotonicity and proof of inclusion are obtained by comparing the root hash of the Merkle Tree database storing contract state with the hash found in the block header announcement that is broadcast. If a client's local database was compromised, it would notice the hash mismatch.

The proof of non-existance is also simple: if a client does not find an object in its local database and the hash of the database matches the latest block header, then it can believe that the object does not exist. The problem is simpler because everyone has the full database. There are protocols that allow a blockchain client to prove that some state exists or does not exist to another client that does not possess the full database [48] but we don't address those here.

Non-equivocation is one of the core problems that a blockchain solves. In the blockchain context, equivocation is called *forking*, where there exists more than one internally consistent view of the blockchain. This is solved in Ethereum by making blocks hard to create, through proof of work. An attacker would need to have enormous computational power to be able to maintain an alternate view of the blockchain without falling behind in block generation. A detailed explanation of the GHOST protocol used by Ethereum to make forks harder to maintain can be found in [102].

Advanced Blockchain Functionality

Although the simple contract above can meet the six operational requirements, while providing the four proof classes, blockchain smart contracts can provide additional functionality. At the time that

the specifications of WAVE 2 were drafted, we assumed that the system would be implemented on a blockchain, so we included functionality that improves system usability but would be difficult to achieve without a blockchain.

Globally unique entity aliases

The blockchain-based storage provider offers globally unique immutable *aliases* in one flat namespace that are resolvable by all entities. This system can be used to bind human readable names to entities, so that attestations can be granted to simple-to-remember names rather than long strings encoding public keys.

This global immutable naming is useful, in that using an alias is as secure as using the full public key. There is no way an attacker can re-bind an alias to trick a user into granting permissions to a different entity, as the alias is immutable. This is different from, say, DNS where a domain name can be changed by an attacker if they compromise the right accounts.

Evaluation of authorization within a smart contract

In WAVE version 2 the entity and attestation objects are not encrypted (Chapter 5 is not integrated). This allows the system to perform additional verification of the objects it is storing. There are two good reasons to do this. The first is that the blockchain state is kept clean: attackers cannot push arbitrary data into the contract. The clients that are watching for new data in the contract, using the Ethereum event system, only need to process valid objects rather than having to filter out bad data. The second reason is that the validity of the objects (entities, attestations and proofs) is known within the contract and can be used by other contracts. This allows the myriad of smart contract use cases (escrow, tokens, autonomous organizations etc.) to interact with WAVE permissions.

For example, because the contract verified that, for each new attestation, the granting and receiving entities are both valid and unexpired, other contracts can take actions based on the validity of that attestation. Further, as the contract supports verifying chains of attestations, we can create logic that requires proof of permissions to execute changes in blockchain state (such as releasing funds from escrow).

A typical scenario might be something like, “Alice will pay Bob 20 Ether in exchange for read permissions on `bob/creditdata`”. To prevent either party cheating, one could put the 20 Ether in escrow, and when the permissions are granted, the contract automatically transfers the funds. This process has a little added complexity, however: transactions are broadcast in the network so it is possible for an attacker to intercept the attestation-publish transaction and then submit their own transaction containing the same attestation. If the replacement transaction is mined first, the attacker would claim the money in the escrow. There are multiple solutions to this, but we opted for a generic one we named *patents*. Bob will create a patent for the hash of the attestation first, which binds the hash to his Ethereum account. Then when the escrow contract observes a transaction posting the attestation that Alice is paying money for, it sends the money to whoever holds the patent for the attestation, not the issuer of the transaction publishing the attestation. This

mechanism can be used for any money-for-data scenario, not just attestations, but the buying of permissions scenario above was the main motivation.

Blockchain Contracts

There are two contracts in WAVE 2. The first implements the alias infrastructure, and the second stores the attestations and entities along with patents.

The alias contract is quite simple:

```
contract Alias {
  /* The alias database */
  mapping (uint256 => bytes32) public DB;
  mapping (bytes32 => uint256) public AliasFor;

  /* The cost of registering an alias, multiplied by the gas price */
  uint256 public AliasPrice;

  /* Alias creation event */
  event AliasCreated(uint256 indexed key, bytes32 indexed value);

  function Alias(uint256 aliasPrice) {
    AliasPrice = aliasPrice;
  }

  function SetAlias(uint256 k, bytes32 v)
  {
    /* Need to pay for alias creation */
    if (msg.value < AliasPrice*tx.gasprice) {
      throw;
    }
    /* Cannot reassign aliases */
    if (DB[k] != 0x0) {
      throw;
    }
    /* Reverse lookup */
    if (AliasFor[v] == 0x0) {
      AliasFor[v] = k;
    }
    DB[k] = v;
    AliasCreated(k, v);
  }
}
```

The contract allows for mapping a 32 byte human readable string to an arbitrary 32 byte value, usually an entity public key. We introduce a cost to dissuade users from "squatting" on multiple names. To deal with changes in the value of Ether over time, we express the cost of an alias as a multiple of the gas price. The gas price will theoretically fluctuate with the value of Ether.

The first alias to bind a key also gets the reverse lookup stored so that you can resolve a key to a name. This feature has some unfortunate security implications as, in our experience, users tend to trust a key based on what it was named, even though names are arbitrary and can be bound by anyone. This is one of the reasons that WAVE 3 uses hierarchical names (which do convey trust) instead of a flat namespace.

The contract for storing entities and attestations in WAVE 2 is listed below. You will notice that it uses functions in an external contract named `bw`. This contract is described in the next section.

The contract begins with a declaration of the events that are emitted by the contract. These events appear in the bloom filter in each block header, allowing clients to efficiently determine if a block contains one of these events. The events are predominantly used for flushing caches, but are also used by publishing entries to know that the transaction they posted has been successfully mined.

```
contract Registry {

    modifier onlyadmin { if (msg.sender != admin) throw; _ }
    address public admin;

    /* Events so that clients can follow interactions with the registry */
    event NewAttestation(bytes32 indexed hash, bytes object);
    event NewEntity(bytes32 indexed vk, bytes object);
    event NewChain(bytes32 indexed hash, bytes object);
    event NewAttestationRevocation(bytes32 indexed hash, bytes object);
    event NewEntityRevocation(bytes32 indexed vk, bytes object);
    event NewRevocationBounty(bytes32 indexed key, uint newValue);
```

We then define the structs used for storing attestations and entities. In contrast with the simple contract above, we also store the *Validity* of the attestations and entities which is accessible from a contract.

```
    enum Validity { Unknown, Valid, Expired, Revoked }

    /* The state of the registry */
    struct EntityEntry
    {
      bytes blob;
      Validity validity;
      uint expiry;
    }
    struct AttestationEntry
    {
      bytes blob;
      Validity validity;
      uint expiry;
    }
    struct ChainEntry
    {
      bytes blob;
      Validity validity;
      uint since;
    }
```

Next we declare the actual variables that store state. These are global variables, but because transactions are applied serially, there are no race conditions. We store full proofs in the contract as well (ChainEntry). This allows contracts to reason about the permissions of entities rather than just individual attestations. The `AttFromVK` stores the list of attestations granted from an entity (entities are identitied by Verifying Key in WAVE 2). This plays a similar role to the queues in the minimal contract listed earlier except that the edge is from issuer to subject as in WAVE 2 the graph discovery is done by walking from the namespace to the proving entity (forwards).

```
    mapping (bytes32 => EntityEntry) public Entities;
    mapping (bytes32 => AttestationEntry) public Attestations;
    mapping (bytes32 => ChainEntry) public Chains;
    mapping (bytes32 => uint) public RevocationBounties;
    /* fromVK -> []atthash */
    mapping (bytes32 => bytes32[]) public AttFromVK;
```

We envisioned a scenario where one might want to revoke an entity or attestation but no longer have any Ether to pay for the transaction. To mitigate this, we allow entities to place a *bounty* in advance that is paid to whoever revokes the given entity or attestation. If this bounty exceeds the transaction costs, then an entity wanting to revoke an attestation could broadcast the revocation object and just wait for a *bounty hunter* to publish it for them. These bounty hunters are incentivized to "help out" because they make more money by publishing the revocation than they spend on the transaction. This scheme is basically a way of pre-paying for the revocation of an entity or attestation so that when it needs to be revoked you do not need to worry about transaction fees.

```
/* Revocation Infrastructure */
function AddRevocationBounty(bytes32 hash)
{
  /* This would be silly to do if there was already a revocation... */
  RevocationBounties[hash] += msg.value;
  NewRevocationBounty(hash, RevocationBounties[hash]);
}
```

Next we have support for adding attestations. Note that unlike the minimal contract, this checks in detail that the attestation is valid, including both the source and destination entities:

```
/* AddAttestation will add an attestation but only if it is valid,
   and the entities it refers to are also valid and in the registry. */
function AddAttestation(bytes content)
{
  var (valid, numrevokers, ispermission, expiry, srcvk, dstvk, hash) =
    bw(0x28589).UnpackAttestation(content);

  /* Even if Att. is invalid, we keep their money so may as well record,
     even if we are assigning to zero hash */
  RevocationBounties[hash] += msg.value;
  NewRevocationBounty(hash, RevocationBounties[hash]);

  /* Next, stop if the Att. was invalid */
  if (!valid) {
    return;
  }

  /* Check the entity expiries */
  CheckEntity(srcvk);
  CheckEntity(dstvk);

  /* Stop if the entities are not ok */
  if (Entities[srcvk].validity != Validity.Valid ||
      Entities[dstvk].validity != Validity.Valid) {
    return;
  }

  /* Stop if Att. is known */
  if (Attestations[hash].validity != Validity.Unknown) {
    return;
  }

  /* Find the min expiry for the Attestation */
  uint minExpiry = expiry;
  if (minExpiry == 0 || (Entities[srcvk].expiry != 0 && Entities[srcvk].expiry < minExpiry)) {
    minExpiry = Entities[srcvk].expiry;
  }
  if (minExpiry == 0 || (Entities[dstvk].expiry != 0 && Entities[dstvk].expiry < minExpiry)) {
    minExpiry = Entities[dstvk].expiry;
  }
```

```
  /* Put it in */
  if (minExpiry == 0 || minExpiry > block.timestamp) {
    Attestations[hash].blob = content;
    Attestations[hash].expiry = minExpiry;
    AttestationFromVK[srcvk].push(hash);
    Attestations[hash].validity = Validity.Valid;
    NewAttestation(hash, content);
  }
}
```

Similarly we have the function for adding entities to the registry. This also makes use of the `bw` contract to unpack the object and validate it.

```
function AddEntity(bytes content)
{
  var (valid, numrevokers, expiry, vk) = bw(0x28589).UnpackEntity(content);

  /* Even if Entity is invalid, we keep their money so may as well record,
     even if we are assigning to zero hash */
  RevocationBounties[vk] += msg.value;
  NewRevocationBounty(vk, RevocationBounties[vk]);

  /* Next, stop if the Entity was invalid */
  if (!valid) {
    return;
  }

  /* Stop if the Entity is known */
  if (Entities[vk].validity != Validity.Unknown) {
    return;
  }

  /* Put it in */
  if (expiry == 0 || expiry > block.timestamp) {
    Entities[vk].blob = content;
    Entities[vk].validity = Validity.Valid;
    Entities[vk].expiry = expiry;
    NewEntity(vk, content);
  }
}
```

There is no way to have a time triggered event run on the blockchain, so if an entity or attestation expires, the validity associated with that object must be updated as part of a transaction. These functions check that an attestation or entity is still valid and update the state. They are called as part of in-contract verification.

```
/* This will update the entity state to expired if it has expired */
function CheckEntity(bytes32 vk) {
  if (Entities[vk].validity != Validity.Valid) {
    return;
  }
  if (Entities[vk].expiry != 0 && Entities[vk].expiry < block.timestamp) {
    Entities[vk].validity = Validity.Expired;
  }
}

/* This will update the att. state to expired if it has expired */
/* We don't do revocation because it's pretty expensive and
   there is ample motivation for it to happen elsewhere */
function CheckAttestation(bytes32 hash) {
  if (Attestations[hash].validity != Validity.Valid) {
```

```
    return;
  }
  if (Attestations[hash].expiry != 0 && Attestations[hash].expiry < block.timestamp) {
    Attestations[hash].validity = Validity.Expired;
  }
}
```

In addition to the ability to store entities and attestations in the global storage tier (functionality that is present in both WAVE 2 and WAVE 3), WAVE2 also has the ability to store proofs (chains) in the registry. Validating a proof means validating all of its constituent attestations and entities. We make use of the function `ADCChainGrants` which performs the actual check that the attestations form a valid proof. Among other things this will also validate that the subject of each attestation is the issuer of the subsequent attestation.

```
function AddChain(bytes content)
{
  var (valid, numatts, chainhash) = bw(0x28589).UnpackChain(content);

  /* Even if chain is invalid, we keep their money so may as well record,
     even if we are assigning to zero hash */
  RevocationBounties[chainhash] += msg.value;
  NewRevocationBounty(chainhash, RevocationBounties[chainhash]);

  /* Stop if invalid */
  if (!valid) {
    return;
  }

  /* Stop if the chain is known */
  if (Chains[chainhash].validity != Validity.Unknown) {
    return;
  }

  /* Now we assemble the chain into scratch */
  for (uint8 attidx = 0; attidx < numatts; attidx++) {
    bytes32 atthash = bw(0x28589).GetChainAttestationHash(chainhash, attidx);
    CheckAttestation(atthash);
    if (Attestations[atthash].validity != Validity.Valid) {
      return;
    }
    var (_1,_2,_3,_4,srcvk,dstvk,_5) =
        bw(0x28589).UnpackAttestation(Attestations[atthash].blob);
    CheckEntity(srcvk);
    CheckEntity(dstvk);
    if (Entities[srcvk].validity != Validity.Valid ||
        Entities[dstvk].validity != Validity.Valid) {
      return;
    }
    bw(0x28589).UnpackEntity(Entities[srcvk].blob);
    bw(0x28589).UnpackEntity(Entities[dstvk].blob);
  }

  /* Now validate the full chain */
  uint16 rv = bw(0x28589).ADChainGrants(chainhash, 0x0, 0x0, "");

  /* And put it in */
  if (rv == 200) {
    Chains[chainhash].blob = content;
    Chains[chainhash].validity = Validity.Valid;
    NewChain(chainhash, content);
  }
```

```
}
```

We also support revocations. Note that in WAVE 2 the revocation scheme is more complex than the commitment revocation scheme in WAVE 3 (§3.5). An entity in WAVE 2 has a list of other entities that are allowed to revoke it (called *Delegated Revokers*). The revocation of an entity A is the hash of A signed by one of A's delegated revokers. This means that the contract must check if the revocation object is signed by one of these delegated revokers before processing it.

```
function RevokeEntity(bytes32 target, bytes content)
{
  /* this will be a nop if it's not an entity */
  CheckEntity(target);
  if (Entities[target].validity != Validity.Valid) {
    return;
  }
  var (_1, numrevokers, _2, _3) = bw(0x28589).UnpackEntity(Entities[target].blob);
  var (validsig, rtarget, rvk) = bw(0x28589).UnpackRevocation(content);
  if (!validsig || rtarget != target) {
    return;
  }
  bool validrevoker = (rvk == rtarget);
  if (!validrevoker) {
    for (; numrevokers > 0; numrevokers--) {
      bytes32 allowed_rvk = bw(0x28589).GetEntityDelegatedRevoker(rtarget, numrevokers-1);
      if (allowed_rvk == rvk) {
        validrevoker = true;
      }
    }
  }
  if (!validrevoker) {
    return;
  }
  Entities[target].validity = Validity.Revoked;
  NewEntityRevocation(target, content);
  if (RevocationBounties[target] != 0 && msg.sender.send(RevocationBounties[target])) {
    RevocationBounties[target] = 0;
  }
  return;
}
```

The revocation of an attestation is similar, but we also take the opportunity to update the validity of the issuer and subject entities in contract state.

```
function RevokeAttestation(bytes32 target, bytes content)
{
  CheckAttestation(target);
  if (Attestations[target].validity != Validity.Valid) {
    return;
  }
  bool  validsig;
  uint8  numrevokers;
  bool _bool;
  uint64  _u64;
  bytes32 srcvk;
  bytes32 dstvk;
  bytes32 rtarget;
  (validsig,numrevokers,_bool,_u64,srcvk,dstvk,rtarget) =
    bw(0x28589).UnpackAttestation(Attestations[target].blob);
  CheckEntity(srcvk);
  bw(0x28589).UnpackEntity(Entities[srcvk].blob);
  CheckEntity(dstvk);
```

```
  bw(0x28589).UnpackEntity(Entities[dstvk].blob);
  /* dstvk now means the vk that signed the revocation */
  (validsig, rtarget, dstvk) = bw(0x28589).UnpackRevocation(content);
  if (!validsig || rtarget != target) {
    return;
  }
  bool validrevoker = (srcvk == dstvk);
  /* rtarget now means a VK allowed to revoke */
  if (!validrevoker) {
    for (;numrevokers>0; numrevokers--) {
      rtarget = bw(0x28589).GetAttDelegatedRevoker(target, numrevokers-1);
      if (rtarget == dstvk) {
        validrevoker = true;
      }
    }
  }
  if (!validrevoker) {
    return;
  }
  Attestations[target].validity = Validity.Revoked;
  NewAttestationRevocation(target, content);
  if (RevocationBounties[target] != 0 && msg.sender.send(RevocationBounties[target])) {
    RevocationBounties[target] = 0;
  }
}
```

The above functions implement the core functionality of the WAVE 2 contracts. We now look at the parts of the contracts that implement additional functionality. The following is the constructor for the contract. It hardcodes the cost of patents and the duration (in blocks) that they remain valid for:

```
function Registry() {
  PatentPrice = 10 ether;
  PatentDuration = 100;
  admin = msg.sender;
}
```

Finally, we have the patent infrastructure that allows escrow contracts to be constructed for the exchange of permissions. This appears in the main registry contract because there needs to be one "authority" for patents that is known to all parties in order for the mechanism to be effective.

```
/* Patent infrastructure */
uint public PatentPrice;
uint public PatentDuration;
event Patent(bytes32 indexed hash);
mapping (bytes32 => address) public Patents;
mapping (bytes32 => uint) public PatentExpiry;

/* Be careful to give the contract extra money if there are
   outstanding patents */
function SetPatentProperties(uint price, uint duration)
  onlyadmin
{
  PatentPrice = price;
  PatentDuration = duration;
}

/* These are the Registry functions */
function NewPatent(bytes32 hash)
{
  /* Pay up */
```

```
    if (msg.value < PatentPrice) throw;
    /* Still a valid patent there bud */
    if (PatentExpiry[hash] != 0x0 && block.number < PatentExpiry[hash]) throw;
    Patents[hash] = msg.sender;
    PatentExpiry[hash] = block.number + PatentDuration;
    Patent(hash);
  }
  function WhoHoldsPatentFor(bytes32 hash) returns (address)
  {
    if (Patents[hash] == 0) return 0;
    if (block.number >= PatentExpiry[hash]) {
      Patents[hash] = 0;
      PatentExpiry[hash] = 0;
      return 0;
    }
    return Patents[hash];
  }
  function ClosePatent(bytes32 hash)
  {
    /* Can only claim your own patent */
    if (Patents[hash] != msg.sender) throw;
    /* Can only claim unexpired patents */
    if (block.number >= PatentExpiry[hash]) {
      Patents[hash] = 0;
      PatentExpiry[hash] = 0;
      return;
    }
    /* Can only claim a patent refund if you put your
       stuff in public domain */
    if (Attestations[hash].validity == Validity.Unknown &&
        Entities[hash].validity == Validity.Unknown &&
        Chains[hash].validity == Validity.Unknown)
    {
      /* Not in public domain */
      return;
    }
    Patents[hash] = 0x0;
    PatentExpiry[hash] = 0x0;
    /* What a good guy! You shared your stuff! Get some money */
    if (!msg.sender.send(PatentPrice))
      throw;

    /* Note that patents do not cover revocations because
       you can use a bounty for that */
  }
}
```

Pre-compiled Contracts

Although the attestations in WAVE 2 are not encrypted, validating within the contract requires validating an Ed25519 signature, among other operations. Early experimentation revealed that implementing Ed25519 in Solidity [49] directly would use too much *gas* (Ethereum's mechanism for limiting the run-time of contracts). To mitigate this, we extended the Ethereum Virtual Machine (EVM) with a *pre-compiled contract* that enables contracts to call WAVE specific functions that are implemented in native code. The precompiled contract chooses the gas cost for the operations it exports, so is free to choose something affordable.

The contract begins with signature verification and a utility function:

```
library bw {

  /* returns true if valid, false otherwise
  */
  function VerifyEd25519Packed(bytes blob) returns (bool) {}

  /* returns true if valid, false otherwise
  */
  function VerifyEd25519(bytes32 vk, bytes sig, bytes body) returns (bool) {}

 /* Returns the 32 byte slice from the given offset
  */
  function SliceByte32(bytes blob, uint32 offset)
  returns (bytes32) {}
```

In addition to the signature verification, the gas cost of manipulating the packed binary structures of attestations, entities, and attestations also proved to be high. There was also the problem that return values could not be variable size in Solidity. To mitigate this, the EVM was modified to contain a *scratch space*. Whenever an entity or attestation was inspected, it was stored in this scratch space. The scratch space only persists for the duration of a transaction, but it allows for contracts to refer to the same attestation or entity multiple times using just the hash, rather than passing the full serialization each time that would consume excess gas. These functions unpack WAVE objects but also place the object in scratch:

```
  /* returns (bool valid, uint8 numrevokers, bool ispermission,
  * uint64 expiry, bytes32 srcvk, bytes32 dstvk, bytes32 dothash)
  */
  function UnpackAttestation(bytes dot)
  returns (bool, uint8, bool, uint64, bytes32, bytes32, bytes32) {}

  /* returns (bool valid, uint8 numrevokers, uint64 expiry, bytes32 vk)
  */
  function UnpackEntity(bytes entity)
  returns (bool valid, uint8 numrevokers, uint64 expiry, bytes32 vk) {}

 /* returns (bool valid, uint8 numdots, bytes32 chainhash)
  * obj len must be a multiple of 32
  * Also puts the dchain in scratch
  */
  function UnpackAccessDChain(bytes obj)
  returns (bool valid, uint8 numdots, bytes32 chainhash) {}

 /* returns (bool valid, bytes32 target, bytes32 vk)
  */
  function UnpackRevocation(bytes blob)
  returns (bool valid, bytes32 target, bytes32 vk) {}
```

These functions allow for extracting the variable-sized attributes of entities, attestations, and chains that could not be returned from Unpack functions. They all take an index so the caller can iterate over the variable sized data. The Unpack function must be called first to place the object in scratch.

```
  /* returns the delegated revoker at the specific index
  * The att must have been unpacked within the calling contract
  */
  function GetAttDelegatedRevoker(bytes32 dothash, uint8 index)
  returns (bytes32) {}

  /* returns the delegated revoker at the specific index
```

```
  * The Entity must have been unpacked within the calling contract
  */
  function GetEntityDelegatedRevoker(bytes32 vk, uint8 index)
  returns (bytes32) {}

  /* returns the hash of the attestation at the give index
  * in the dchain. att must be in scratch
  */
  function GetDChainAttestationHash(bytes32 chainhash, uint8 index)
  returns (bytes32 atthash) {}

 /* GetAttNumRevokableHashes(bytes32 atthash)
  * Gets the total number of vulnerable hashes for the given att
  * att must be in scratch
  */
  function GetAttNumRevokableHashes(bytes32 dothash)
  returns (uint32) {}

 /* GetAttRevokableHash(bytes32 atthash, uint32 index)
  * Gets the given revokable hash (entity or att) that affects
  * the attestation.
  */
  function GetAttRevokableHash(bytes32 atthash, uint32 index)
  returns (bytes32) {}

 /* GetChainNumRevokableHashes(bytes32 chainhash)
  * Gets the total number of vulnerable hashes for the given
  * chain of attestations. The chain and all attestations must
  * be in scratch.
  */
  function GetChainNumRevokableHashes(bytes32 chainhash)
  returns (uint32) {}

 /* GetChainRevokableHash(bytes32 chainhash, uint32 index)
  * Get a specific revokable hash for the chain (att, entity)
  */
  function GetChainRevokableHash(bytes32 chainhash, uint32 index)
  returns (bytes32) {}
```

Finally we have the function that validates an entire proof. This allows clients such as other smart contracts to validate a proof without including their own copy of code to do so. The constituent entities and attestations must have been unpacked in the same transaction in order for this to fully verify, otherwise it just verifies what it can.

Permissions in WAVE 2 are simple: the `adps` field is a bitmask of the required permissions, including publish, query, subscribe and subscribe with + or $\star$ fields. There are additional permissions not covered here but they were never used.

```
 /* rv = 200 if chain is valid, and all dots are valid and unexpired and
  *          it grants a superset of the passed adps, mvk and suffix
  *          and all the entities are known to be unexpired
  * rv = 201 same as above, but some entities were not present in Scratch
  * else  a BWStatus code that something went wrong
  */
  function AChainGrants(bytes32 chainhash, bytes8 adps, bytes32 namespace, bytes urisuffix)
  returns (uint16 bwstatus) {}
}
```

Normally, when a contract is accessed by another contract, it is referred to by its address, so you might see `OtherContract(address).function()`. The WAVE precompiled contract is accessed in the same way, except that when the EVM sees an address in its list of precompiled

contracts, it executes the native code added to the EVM, rather than executing instructions from the blockchain state. The address for our contract was 28589, as can be seen in the registry contract.

4.4 Blockchain Scalability Concerns

Although the blockchain is capable of implementing the fundamental storage requirements and even adding functionality that is quite useful, blockchains at present have very limited throughput. We spent a considerable amount of time evaluating a blockchain for storage of the authorization objects before determining that it would not scale past a few tens of transactions per second. Even though the blockchain is known to be expensive, the rationale for a blockchain in the first place was that authorization changes are typically more rare than data operations, and we only need to put authorization data on the blockchain. Ultimately, however, the scalability concerns proved onerous enough to warrant consideration of alternate approaches. This section details the study that showed blockchains would not scale sufficiently.

WAVE 2 has *agents* and *miners*. The agent handles the application API, such as forming and verifying proofs. An agent incorporates an Ethereum blockchain client. The miner is just a blockchain client and performs the tasks required to keep an Ethereum blockchain progressing, e.g. mining transactions into blocks and solving proof of work problems.

A primary concern in using a blockchain is the cost of participating in the chain. We perform a large scale test on over 100 WAVE nodes and emulate different levels of load on the blockchain to observe the impact on CPU, memory, and network bandwidth consumption of these nodes. We monitor the current "age" of the chain on each node. The age is a good indicator of general health as any problems with the blockchain will cause it to fall behind in processing new blocks, increasing the age. If an agent cannot keep up with chain updates, it must treat proofs as invalid as it is potentially unaware of recent revocations.

Experimental Setup We run WAVE on three different platforms. Docker containers are used to encapsulate 100 nodes, consisting of both miners and agents, on Amazon EC2. We ran seven `m4.16xlarge` instances featuring Intel Xeon CPUs and SSD-backed storage that guarantee a baseline performance of about 1000 IOPS. We randomly distribute the containers among these hosts, such that each host has at least 4 CPU cores and 8 GB of memory per container. `netem` shapes each container's network latency and bandwidth. A container is assigned to one of four networking classes based on a profile of wired Internet connections in North America [15]. The breakdown of nodes among these different constraint classes is given in Table 4.1. In addition to the AMD64 cloud nodes, three agents are run on Raspberry Pis with quad-core armv7l CPUs and 1 GB of RAM and three agents are run on i386 machines with Intel Atom CPUs and 2 GB of RAM. Peers refers to how many connections are opened to other machines to transfer blockchain state. More peers should theoretically allow a node to receive updates to the blockchain faster but also requires more bandwidth.

We study the performance of the WAVE agents and miners during four phases of operation.

	Net class [speed Mbps/latency ms]			
Role	100/3 A	17.2/5 B	2/30 C	1/100 D
Miner - 20 peers	10			
Agent - 20 peers	12	11	11	11
Agent - 2 peers	12	11	11	11

Table 4.1: Breakdown of AMD64 cloud nodes in testbed.

1. *Fast Sync* [105] occurs when a new node is brought online. The node must synchronize with the blockchain to reach a consensus on WAVE's current state.

2. *Idle* is a period of minimal blockchain activity. Although blocks contain no transactions, new blocks are mined. Agents must process these new blocks to remain synchronized with the chain.

3. *Attack* is a period of intense activity, effectively at the level of a concerted attack on the chain by a party with infinite funds. This is compounded by WAVE's blocks becoming temporarily enlarged, accommodating more than 130 attestation objects per block, meaning each block consumes 25M gas, an order of magnitude more than the gas limit of Ethereum's main chain (roughly 4M).

4. *Normal* is a period of typical city-scale load on the blockchain. The load was constructed to mirror the expected authorization load associated with IoT devices arising from a city the size of San Francisco. The rationale for the exact parameters of this load is omitted here, because it is not particularly contentious to state that the load on a global authorization system would eventually surpass hundreds of transactions per second. This particular load is sized large enough to demonstrate the scaling characteristics of a blockchain.

Table 4.2 summarizes the **CPU** measurements collected for all platforms. No WAVE node was CPU-bound at any point. Miners use multiple CPU cores regardless of the load on the chain, as expected. CPU consumption for regular agents was higher when the blockchain was under attack, but never exceeded roughly half a core.

WAVE agents and miners opportunistically take advantage of extra **memory** available to them for caching, but do not require this to function. We observe no issues on machines with at least 2 GB of memory. With the Raspberry Pi's 1 GB of memory, we observed several out-of-memory conditions on their WAVE agents when the blockchain load was at attack levels. The Raspberry Pi agents had sufficient memory to participate whenever the chain was idle or under normal city-scale load.

The **bandwidth** differences across Internet speeds are not statistically significant when the agent is caught up on the chain. UDP bandwidth used for peer discovery is not significant compared to the bandwidth used for transferring state and is omitted. Table 4.3 shows the average bandwidth

Type	Peers	Idle (1h)	Normal (30h)	Attack (4h)
M,x10,x64	20	3.80±0.72	3.96±0.26	4.84±0.75
A,x45,x64	20	0.04±0.01	0.03±0.01	0.59±0.30
A,x45,x64	2	0.05±0.01	0.03±0.01	0.48±0.32
A,x1,armv7	20	0.46±0.08		1.39±0.10
A,x2,armv7	2	0.15±0.01		1.58±0.11
A,x1,atom	20	1.22±0.14		1.37±0.09
A,x1,atom	2	0.47±0.06		0.61±0.10

Table 4.2: CPU utilization as [number of cores] for blockchain participation. The first column indicates the role (Agent or Miner), number of nodes and architecture for each node type.

Type	↑↓	Attack (1h)	Idle (30h)	Normal (4h)
M,10,20	in	103±6	2.74±1.10	2.19±0.15
	out	139±17	5.66±17.7	2.53±0.46
A,45,20	in	114±12	2.89±0.52	2.69±0.93
	out	140±22	2.93±1.33	11.85±35.2
A,45,2	in	15.4±1.9	0.85±0.36	0.54±0.16
	out	9.7±2.1	0.79±1.14	1.24±2.52

Table 4.3: Bandwidth [KiB/s] for blockchain participation. The first column indicates the role (Agent or Miner), quantity, and peer count for each node type.

used by agents at different levels of chain activity. The bandwidth used to participate in the chain is reasonable even during periods of excessive load.

The bandwidth is impacted by the number of peers (connections to other machines to receive/-transmit blockchain state), which is likely indicative of an implementation flaw in the Ethereum golang client, as having more peers should not cause a participant to download significantly more data. All additional downstream traffic at attack load in the 20-peer case is redundant, so at least 100KiB/s of 114KiB/s is data that need not be fetched from peers. Nevertheless, participation in a chain that is under a DoS attack only requires a modest amount of bandwidth.

Chain age is the number of blocks separating the head of the blockchain from the latest block known to a node. Nodes stay up to date on the chain during idle and normal period, independent of peer count and available network bandwidth. We never observe a node falling behind by more than nine blocks during idle and normal periods, equivalent to maximum staleness of about two minutes.

Figure 4.1 shows the age of the chain for each of the 100 EC2-based WAVE nodes across our emulated attack. All nodes are caught up to the head of the chain until activity dramatically increases at 04:30 UTC. At that point most nodes fall one or two blocks behind, but quickly recover.

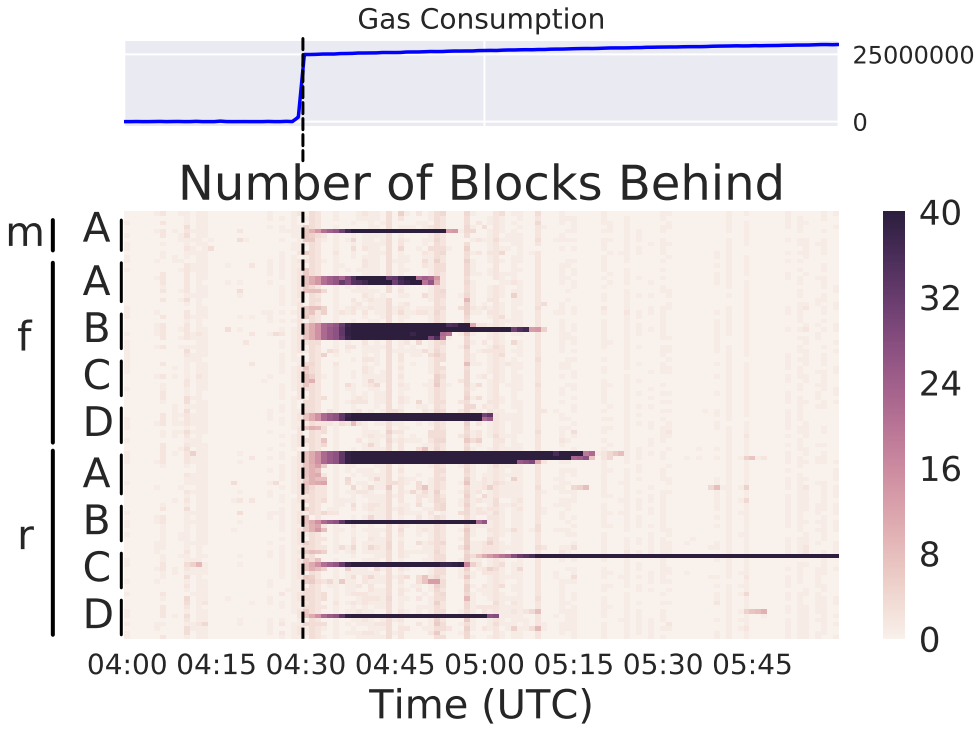


Figure 4.1: Number of blocks behind the head of the chain over time (x) for each node. The y axis breaks nodes down by role: miner (m), agent with a full set of 20 peers (f), and agent restricted to two peers (r) as well as by the net classes defined in Table 4.1.

A handful fall more behind and require about 30 minutes to recover. Upon further investigation, these nodes were found to be running on the same EC2 host, and we concluded that the instance was suffering from disk IOPS saturation. 45 minutes into the attack, however, nearly all agents are again caught up to the head of the blockchain.

Conclusion At the time these experiments were performed, the goal was to construct a system that operated at the scale of a city. As our goals became more ambitious, targeting a system that could handle authorization for the world, it became evident that an Ethereum blockchain as currently implemented could not support a load greater than our attack stimulus of 130 created attestations per block (less than 10 per second), far below realistic estimates of the load generated by a global system. While there are ongoing attempts to scale blockchains through sharding [116] or off-chain transactions [94], none of these have made significant progress over the course of the WAVE research (3 years). This is what led us to construct the ULDM storage tier presented below.

4.5 Unequivocable Log Derived Map

Whilst the blockchain storage tier provides useful features. We do not need all the features of a blockchain. The minimal feature that is required is the ability to maintain a key value store that can prove its integrity. This section discusses the design of system that builds on this idea.

WAVE version 3 implements the scheme in §4.2 without the additional functionality of aliases or patents as implemented in WAVE 2. We implement this key-value based storage scheme while meeting the security requirements by utilizing a cryptographically verifiable data structure that can issue the required integrity proofs.

The ULDM API consists of four functions: Get and Put are used for placing/retrieving entities, attestations, name declarations (§3.9) and revocation secrets (§3.5) in storage; Enqueue places an object hash at the end of a named queue, and IterQueue allows retrieval from a queue. The queue functions implement the Notify functions above that facilitate discovery, allowing an entity to notify another entity that a new attestation has been granted to them or a new name declaration has been published.

Removing trust in the server operator through the use of cryptographically verifiable data structures is not new, there are several projects that have similar goals, such as Certificate Transparency (CT) [82] or Key Transparency (KT) [64]. Although it seems like the requirements could be met by using CT or KT, neither of those are appropriate. CT provides a log which is capable of inclusion proofs, but cannot form a proof of non-existence so does not allow for WAVE's revocation scheme. KT provides the necessary proofs, but the security of the datastructure relies on a very heavy-weight auditing scheme that would perform poorly in our context. Briefly, KT requires that every time the map changes (an *epoch*) every user checks that all the objects they have ever placed in storage are still there and have not been modified. KT was designed for a small number of objects per user (≈ 1) and infrequent epochs. WAVE expects thousands of objects per user, frequent epochs, and additionally that entities do not have a persistent online presence capable of performing this kind of auditing.

Instead, we propose an *Unequivocable Log Derived Map (ULDM)*, a transparency log based on the Verifiable Log Backed Map (VLBM) [44]. A VLBM allows the storage server to form proofs of integrity. The VLBM whitepaper is brief and incomplete: it does not discuss auditing, such as which proofs are exchanged or how they are published, so it is unclear how the VLBM prevents equivocation (i.e., presenting different internally consistent views to different clients). To our knowledge, there is no complete open-source VLBM implementation (the code in the repository [68] only implements a subset of the paper, omitting the log of map roots), so we could not build upon the VLBM or infer its scheme from the code. The ULDM is our approach to filling in the missing pieces, such as an auditing scheme to prevent equivocation and secure batching to increase performance.

A ULDM is constructed using *three* Merkle trees, each serving a different purpose, as shown in Fig. 4.2. The first tree is the Operation Log, which stores every Put and Enqueue operation and can prove the log is append-only. These operations are then processed in batches into the second tree, the Object Map. This is used to satisfy queries and prove that objects exist or do not exist within the map. The ULDM Object Map is different from [44] as it only stores the hashes of the

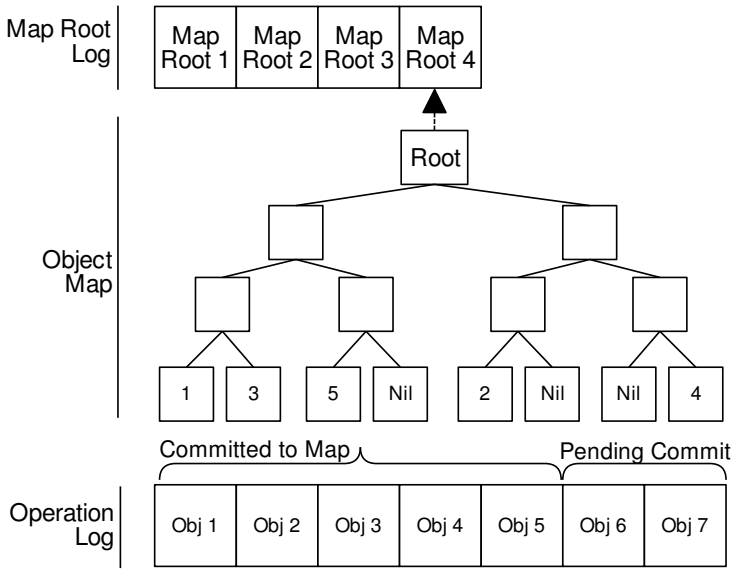


Figure 4.2: An Unequivocable Log Derived Map (ULDM) built from two Merkle tree logs and a Merkle tree map

objects. Finally, every map root created when a batch is processed is inserted into the third Merkle tree, the Map Root Log. This makes the data structure efficiently auditable, as we discuss in §4.5.

Inserting Values

To insert a value, the ULDM server: (1) Inserts the value into the Operation Log. (2) Creates a new version of the Object Map with the new entries from the Operation Log. (3) Inserts the new map root into the Map Root Log. Step 1 is batched (multiple values are inserted into the Operation Log together) as is step 2 (multiple values are inserted into the Object Map together). Step 3 is synchronous with step 2.

Merge Promises

Inserts would ideally be performed synchronously, allowing the server to return inclusion proofs for all three trees in response to the insert. Unfortunately, this results in a severe performance penalty as the ratio of new data to overhead (internal nodes in the trees) is poor. This is the same conclusion that Certificate Transparency reaches, and we use a similar solution: batching with promises. When inserting a value, a client receives a *merge promise* (called Signed Certificate Timestamp in CT) which states that the inserted value will be present by a certain point in time. In addition to the absolute timestamp used in CT, ULDM merge promises include the version of the root log as this allows a proof of misbehavior without a trusted source of time. Uncompromised clients must check the value has been merged later. To prove misbehavior when a value is not inserted on time, a client can present a merge promise along with a signed Map Root Log head

where the corresponding Object Map does not contain the value and where the version of the head is greater than that in the promise; i.e., a server would need to stop operating completely if it wishes to both avoid merging an object and revealing it is compromised.

Retrieving Values

To retrieve a value, the client sends the storage server the Map Root Log version that it received in a previous request, along with the object identifier it is retrieving (e.g., the hash of an attestation or revocation commitment). If the object exists but has not yet been merged, the merge promise will be returned. There is no guarantee that the storage server will return a value before its merge promise deadline. If the object has been merged or does not exist, the server responds with: (1) the object or nil, (2) a proof that the object existed or did not exist in the Object Map at the latest map root, (3) a proof that the latest map root exists in the Map Root Log at the current Map Root Log head, and (4) a consistency proof that the current Map Root Log head is an append-only extension of the version the client passed in its request. This mechanism allows the client to verify that every map satisfying their queries is contained in the Map Root Log, and that the Map Root Log is consistent. Notably, it does not allow the client to verify that the map was correctly derived from the Operation Log. This task is performed by the auditors.

Auditing

An auditor is a party that connects to a storage server and replays the Operation Log to construct replicas of the Object Map and check the Map Root Log. Each client reports the latest Map Root Log head it obtains from the server (signed by the server along with a version number) to the auditors with some frequency. As the entries in the ULDM object map are the hashes of the objects, not the objects themselves, the map constructed by the auditor is several orders of magnitude smaller than the sum of stored objects. For every entry in the Map Root Log, the auditor will read the incremental additions to the map from the Operation Log and apply them to its own copy. It then ensures the hash of the replica Object Map root matches the hash stored in the Map Root Log, proving that the map is correctly derived from the operation log (no objects were modified or removed).

The strength of the ULDM auditing scheme is that a client can report a single value to an auditor (the client's Map Root Log head) and this is sufficient to catch any dishonesty that might have occurred at any point in the client's history. Without the Map Root Log (such as in [68]), any auditing scheme would need to make the client report every Object Map root to the auditor or take the risk that some dishonesty might remain undiscovered. To see how this might occur, imagine that a storage server removes a revocation from the map, answers a query and then re-adds the revocation. Without the Map Root Log, if the client only reports the final map root to an auditor, it would conclude it is valid. In the ULDM case, the client would report the Map Root Log head which covers all prior map versions, enabling the auditor to discover that the previous query was satisfied from an invalid map.

Detecting dishonesty with a single infrequently-reported value has important scalability implications: as we expect there to be many clients, it is important that the load placed on auditors is much less than the query load generated by the clients, otherwise, only large companies could afford to be auditors. In the ULDM model, it is sufficient for a client to contact an auditor rarely (perhaps once a day) to ensure any prior equivocation is discovered.

We expect clients to periodically check in with a random auditor from a public list of auditors. This ensures that the storage server cannot maintain different states for different auditors as it will be discovered when auditor receives a Map Root Log head from a client that is inconsistent with the one received from the storage server directly.

Security Guarantee

We formalize the security guarantee of a ULDM, as follows. By honest client, we denote a client that is neither faulty nor compromised.

Guarantee 1 (ULDM)**.** *Let C be a set of honest clients and S be a ULDM server. Observe that the Merge Promises following insert requests by these clients and Map Root Log heads sent with retrieval requests by these clients define a partial ordering L over all requests received by S. Suppose that there exists a nonempty set R of requests made by clients in C, such that there exists no possible history of requests made to S that is consistent with both L and all of S's responses to requests in R.* ***If*** *there exists an auditor A such that each client in C has sent A a Map Root Log head it received from S* at least as recent *as the one it received for its latest request in R,* ***then*** *one of the following holds:*

1. *One or more clients in C will be able to detect the inconsistency by inspecting the responses it received to requests that it made to S.*
2. *The auditor A will be able to detect the inconsistency by inspecting the Map Root Log heads it received from clients in C and from S.*

A proof sketch for Guarantee 1 that captures the main points behind a more formal proof is as follows:

Proof Sketch for Guarantee 1. We show that if neither clients in C nor the auditor A detect an attack, then there exists a possible history H of requests consistent with L and all responses to requests in R. Concretely, we show that the Operation Log that the storage server tells the auditor A is such a valid history H. Because A did not detect an inconsistency, we know that, for each client $c \in C$, (1) its Map Root Log head, at some point after its last request in R, is consistent with H. Because c did not detect an inconsistency, we know that (2) c's sequence of Map Root Log heads is append-only, (3) for each request, the returned object did (or did not, if no object was returned) exist in the Object Map, and (4) for each request, the Map Root Log at the time of the request contains the object map used in (3).

Together, (1) and (2) indicate that (5) the client's entire sequence of Map Root Log heads is consistent with H. Together, (3) and (4) indicate that (6) the response received for each request in R is consistent with the current Map Root Log head at the time of the request. Putting together

(5) and (6), we can conclude that the response that each client receives to each request in R is consistent with H. Putting together (2) and (6), we can conclude that H is consistent with the partial ordering imposed by Map Root Log heads for each client c. Because clients make requests to the server to validate every Merge Promise, this also guarantees that H is consistent with the partial ordering imposed by Merge Promises. Thus, H fulfills all desired properties. □

4.6 ULDM Storage Evaluation

Since an entity in WAVE does not communicate with any other entity, except via the storage, the scalability of the solution depends on the performance of the global storage.

The ULDM-based system is shared-nothing and horizontally scalable: the performance of one node does not limit the performance of the overall system. Consequently, we do not need quantitative evaluation to demonstrate that the system could handle an arbitrary load. Nevertheless, we evaluate the performance of a single node to emphasize the increase in performance compared to a blockchain, as a single ULDM node outperforms our characterization of the 100-node Ethereum blockchain-based solution

	PUT 2KB	GET 2KB	Enqueue	Iter Queue
Latency [ms]	10.7	10.4	10.1	10.0

Table 4.4: Average storage operation time (ms/op) under 4 uniform loads ($\approx$ 100 requests per second), measured over 30 seconds ($\approx$ 3k requests per type).

Table 4.4 shows the average latency of the ULDM storage performing single operations at a time (i.e. just GETs or just PUTs). The times for the ULDM-based storage include both the generation of the proofs server-side and the verification of the proofs client-side. Every operation concerns a unique object, so there is no caching.

This ULDM storage was constructed using Trillian [67] backed by MySQL. Fig. 4.3 shows the limits of a single node, where performance for PUTs degrades at approximately 110 requests per second and performance for GETs degrades at approximately 200 requests per second. We expect that performance could be increased if Trillian were deployed on Spanner [35] as the designers intended, but defer this to future work. Note that in this evaluation, every operation concerns a unique object, so as to benchmark the underlying cost of forming proofs, rather than the cache. Real workloads would likely have more cache hits.

Although our storage implementation is unoptimized and built using an off-the-shelf Merkle tree database, single nodes handle insert loads an order of magnitude higher than possible on a blockchain system [37]. In addition, every added node scales the capacity of the system linearly. We envision that multiple storage providers, potentially operated by distinct parties, would operate in parallel, similar to Certificate Transparency [82].

4.7 Storage Through DNS

The blockchain based system in §4.3 and the ULDM system in §4.5 both implement the scheme described in §4.2. Before the development of the requirements in §4.1, WAVE 1 used a simpler storage mechanism based on an existing highly-available small-object store: the Domain Name System.

TXT records in DNS are designed for allowing a name to resolve to a small text field. We can use this to allow for an object hash to be resolved to the contents of the object, for example: `764efa883dda1e11d.wave.io` could resolve to a TXT record containing an Entity object.

There are a few advantages to this system:

Availability: DNS already exists and offers high-availability with replication and local caching. Resolution typically occurs from a server that is close by (with some operating systems performing caching on the end device).

Integrity: DNSSEC provides a means for DNS records to be signed by a chain leading up to a root of trust. This prevents attacks where a compromised intermediate server returns invalid results for a DNS query. This is important for revocations, where an attacker would benefit from claiming a revocation object does not exist.

Unfortunately there are disadvantages to DNS and DNSSEC that led subsequent versions of WAVE to adopt alternate mechanisms:

Reliance on a central authority: DNSSEC relies on a signature chain leading to a central root of trust. If this root, or the intermediate domain key, were to be compromised it would enable an attacker to pretend revocations or attestations do not exist.

Not suited for large objects: TXT records are limited to 255 characters, and while mechanisms exist for stitching together multiple records to form larger objects, stretching this to hundreds of KB does not seem appropriate.

In summary, DNS crudely supports the functional requirements of our storage system (it can implement the key-value scheme in §4.2) but does not meet the security requirements.

4.8 Storage Conclusions

WAVE requires a fairly narrow set of requirements from the storage tier. Effectively GET and PUT with some guarantees about the integrity of the state.

While this is possible to achieve with a blockchain, as demonstrated above, we realized that WAVE does not require the full functionality that a blockchain provides. The key difference is that blockchains have their origins in financial transactions where a set of transactions requires at best causal consistency with a partial ordering and at worst strong consistency with a total ordering. This makes sharding (splitting the blockchain into multiple independent segments) quite challenging. For example, if accounts are hosted on two different shards and a transaction that concerns both accounts takes place, we must ensure that either both shards execute the transaction, or neither of them do, and that the transaction happens in a particular order with respect to other

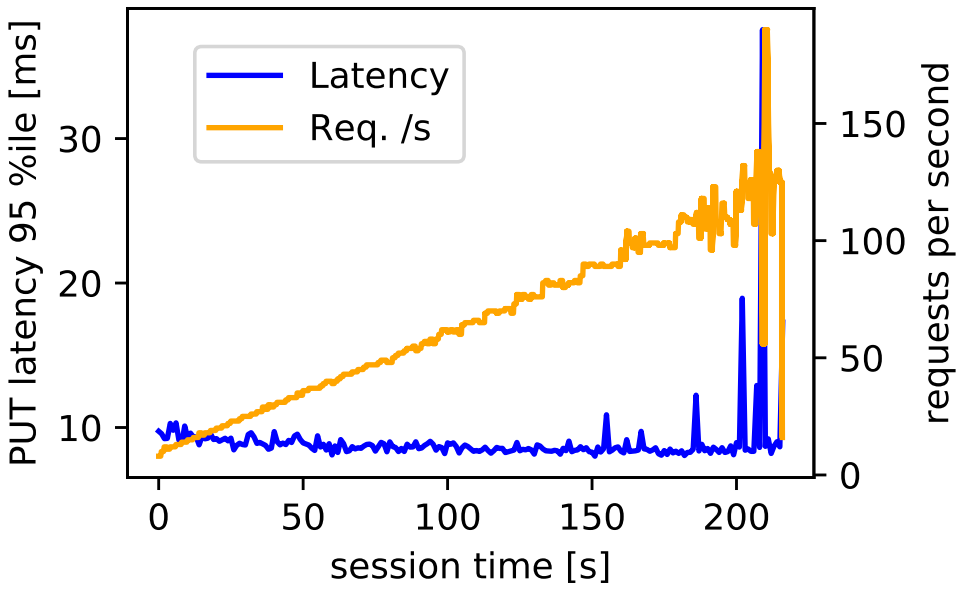
transactions concerning the same accounts. This requires a degree of consensus between all shards, and this consensus is usually the bottleneck.

WAVE does not have the same requirements. The creation of an entity or attestation does not have an ordering requirement with respect to any other entity or attestation. The only ordering requirement introduced is during notifications where there is a total ordering of all notifications for a given entity. This is within a single server, however, so there is no requirement for distributed consensus.

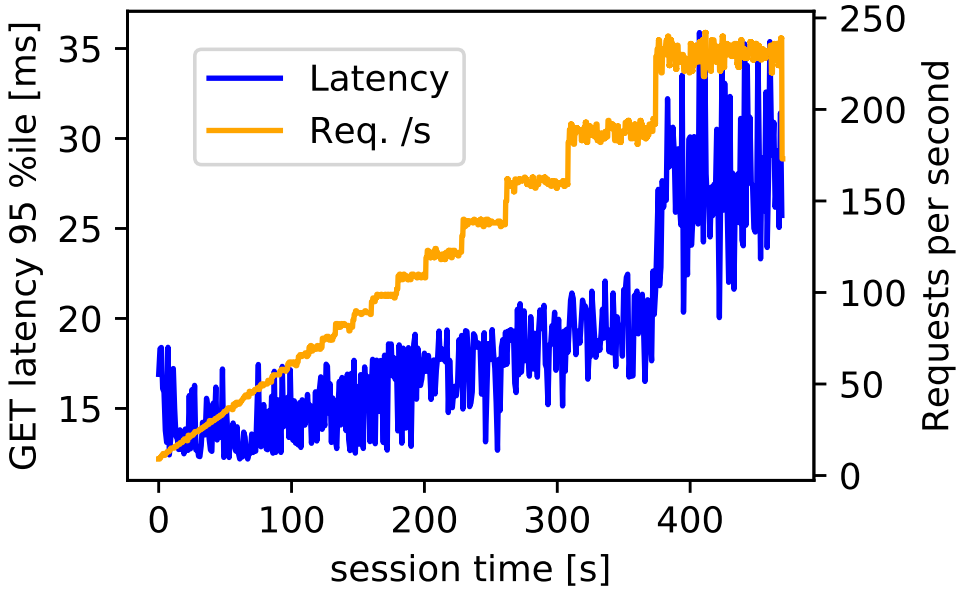
As a result, engineering an arbitrarily horizontally scalable system that meets WAVE's requirements is much easier than engineering an arbitrarily horizontally scalable blockchain. The ULDM storage tier described above has no mechanism for performing consensus between servers. In fact one server need not even know about the existence of other servers.

The ULDM can provide more functionality than we use in WAVE. Smart contracts on Ethereum can be viewed as a mapping of an input transaction onto a sequence of state transitions in the Merkle tree. Such functionality could also exist in a ULDM server, with some restrictions. The ULDM as implemented performs a minimal mapping of "transactions"—it only hashes the object to determine the key or ensures that there has not already been an object stored with a given key. This is not intrinsic, however. We could introduce more feature-rich smart-contract like functionality into the ULDM which would make it useful for a variety of use cases, as long as we preserve the property that a state transition on one server does not have any dependencies on any state transitions on other servers.

Use cases for this functionality would include ensuring that stored objects meet some requirements, such as being well formed or signed by a specific party. Complex schema could be enforced, including relations between objects like foreign key constraints. In the most general case, objects could be treated as parameters to arbitrary turing-complete functions that manipulate state in the local ULDM Merkle tree, as in Ethereum. The only requirement is that all the state that is referenced or manipulated must be local to the ULDM. It is worth noting that this requirement precludes applying the ULDM model to the financial transaction use cases that dominate blockchain research at the moment.



(a) 95th %ile PUT



(b) 95th %ile GET

Figure 4.3: Latencies for ULDM PUT/GET as the throughput is ramped up to the single-node maximum.

Chapter 5

Private Delegation

By building on top of a publicly accessibly storage tier, such as a blockchain or a ULDM, we introduce the requirement for a cryptographic mechanism to maintain the data privacy of the particular objects involved. While the global authorization graph does not contain any sensitive application data itself, the permissions granted between entities are a form of metadata that can reveal more about a person or organization than we would like. For example, when observing the graph, an attacker can infer which devices and services are running within an organization by inspecting the resources that permissions are granted on. Furthermore, the structure of the graph allows an attacker to infer organizational structure by inspecting the delegations between entities.

Our solution to this is to encrypt attestations and name declarations. Unfortunately standard methods of encryption, such as PGP, are insufficient due to one or more of the system requirements. In particular the desire to avoid ordering constraints and to permit functioning with offline participants adds complexity to the encryption solution. For example, if an attestation is granted by B to C and then later, entity B is granted an attestation by A, then C needs to be able to decrypt the new attestation $A \rightarrow B$. Traditional mechanisms, such as B re-encrypting the new attestation under C's key are not appropriate as B may be offline.

To solve these problems, we introduce a new scheme based upon the principle of *reverse discovery*. The premise is that an entity can traverse the global authorization graph backwards from itself, through paths of authorization, to the namespace authorities. With each edge in that graph that is traversed, the entity learns secrets that lets it decrypt further edges, needed to continue the search. That is to say, the secrets that an entity must obtain to decrypt an attestation granted *to* B are contained within attestations granted *from* B. These secrets are collected by the entity performing the discovery procedure (also referred to as the *proving entity*) and the process does not require any other entities to perform additional actions or participate in the procedure in any way, other during the initial attestation grant.

This method allows us to encrypt attestations such that the entities that can use the attestation in a valid proof are capable of decrypting the attestation. If an attestation could not be used in a proof by the proving entity because there is no path from the subject of the attestation to the proving entity, or if the path grants different permissions, then the proving entity cannot learn the policy (i.e., what permissions are granted) nor the issuer (i.e., who created the attestation) of the

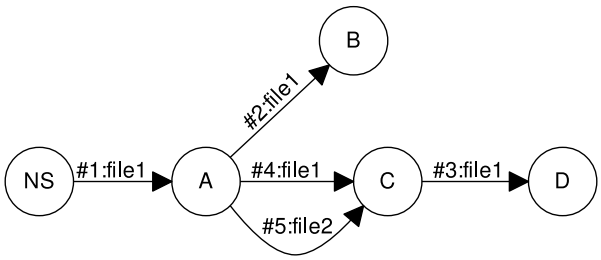


Figure 5.1: The number to the left of each colon indicates when the attestation was created. The string to the right denotes the resource on which it grants permission.

attestation. We name this encryption mechanism *reverse-discoverable encryption* (RDE). It does not require out-of-band communication between entities, and works even if attestations are created out of order.

We present our solution incrementally: §5.1 formalizes the problem that RDE solves. §5.2 presents a simplified design of RDE, based on traditional public-key encryption, that provides a weak but useful security guarantee called "structural security." §5.3 augments the simplified RDE with *policy-aware* encryption to provide a significantly stronger notion of security, at the expense of making discoverability of attestations inefficient. §5.4 presents our final protocol, which provides both efficient discovery of attestations and a significantly stronger guarantee than structural security.

For all the security guarantees stated in this section, and in the appendix, we assume that the attacker Adv is computationally-bounded, and that standard cryptographic assumptions hold.

5.1 Private Delegation Requirements

We formalize the problem in terms of the global authorization graph; an example is shown in Fig. 5.1. For **correctness**, we require that each entity can decrypt all attestations that it can use to form a valid proof where it is the subject. In Fig. 5.1, entity `D` should be able to see attestations `#1`, `#4`, and `#3`. Correctness does *not* require `D` to be able to see attestation `#2`, as there is no path from `B` to `D` granting access to `file1`. Similarly, correctness does *not* require `D` to be able to see attestation `#5`, as there is no path from `C` to `D` granting access to `file2`. For **security**, we would like each entity to see as few additional attestations as possible.

5.2 Structural Reverse Discoverable Encryption

This section explains a simplified (and weaker) version of RDE that is helpful to understand the main idea behind our technique. For this version alone, assume there are no revoked/expired attestations.

Each entity has an additional public-private keypair used only for encrypting/decrypting attestations, separate from the keys used to sign attestations. This keypair is governed by two rules:

when an entity grants an attestation, it (1) attaches the decryption private key to the attestation, and (2) encrypts the attestation, including the attached private key, using the public key of the attestation's subject (recipient). For example, in Fig. 5.1, `#3` contains sk_C and is encrypted under pk_D (i.e., $\text{Enc}(\text{pk}_D; \texttt{\#3}||\text{sk}_C)$).

The included private key in an attestation, say α, when decrypted by a proving entity, allows for the decryption of other attestations granted to the issuer of α. We are simultaneously granting the permissions in α and the ability to decrypt attestations that can be joined with α in the formation of a proof.

This meets the correctness goal; `D` can decrypt `#3` as `#3` is encrypted under pk_D. In decrypting `#3`, it obtains sk_C, which it can use to decrypt `#4`. This works even though attestation `#4` was issued after `#3`. In decrypting `#4`, it obtains sk_A, which it can use to decrypt `#1`. Essentially, each entity can see the attestations it can use in a proof by decrypting them in the reverse order as they would appear in a proof.

This achieves a simple security guarantee called **structural security**, which allows an entity e to see any attestation α for which there exists a path from $\alpha.\mathsf{subject}$ to e. We call it "structural" security because only the structure of the graph, not the policies in attestations, affects whether α is visible to e.

While structural RDE uses traditional public-key encryption, it is quite distinct from systems like PGP. PGP allows a party to encrypt a message that can be decrypted by another party, but it does not include long-lived private keys in the messages for the purposes of enabling future decryption operations. In the PGP world, message re-encryption is typically used when a message needs to be read by a new party. This method does not work in our paradigm because it introduces an ordering constraint (so that upstream attestations are available for inclusion during attestation creation) or requires parties to be on-line to perform re-encryption of upstream attestations created after a given downstream attestation.

Over the next few sections we develop a specific implementation of RDE with improved privacy and performance properties by altering the cryptography and altering the structure of the encrypted objects, but the general principle remains identical: a proving entity obtains secrets contained in an attestation that allows it to decrypt other attestations granted to the issuer.

5.3 Policy-Aware Reverse Discoverable Encryption

Structural security only takes into account the structure of the graph, not the policy of each attestation (i.e., the resources and the expiry). For example, structural RDE allows `D` to decrypt `#5`, though this is not necessary to meet the correctness goal; `D` cannot form a valid proof containing `#5` because its policy differs from `#4`'s (they delegate access to different files). Thus, `D` has visibility beyond what is needed to form valid proofs. With policy-aware RDE, we achieve a stronger notion of security that prevents `D` from decrypting `#5` by making two high-level changes to structural RDE.

First, whereas structural RDE encrypts each attestation A according to only $A.\mathsf{subject}$, policy-aware RDE encrypts each attestation A according to both $A.\mathsf{subject}$ and $A.\mathsf{policy}$. Second, whereas

structural RDE includes a key in A that can decrypt *all* attestations immediately upstream of A, policy-aware RDE includes a key in A that can only decrypt upstream attestations with *policies compatible with* $A.\mathsf{policy}$.

Choosing a suitable encryption scheme.

Because the policy of an attestation determines how it is encrypted, the encryption scheme must be *policy-aware*. If one were to apply traditional public-key encryption, as used in structural RDE, you would need to generate several keys, one for each policy. This scheme introduces an untenable constraint: you can only grant an attestation A to entity E if E has generated the key corresponding to $A.\mathsf{policy}$. This introduces a communication requirement between $A.\mathsf{issuer}$ and E which violates the goal of operating with offline participants. The use of a more powerful encryption scheme, however, can avoid this problem. We use the RTree policy type to explain our policy-aware RDE; policy-aware RDE for general policies is relegated to Appendix A.

We use Wildcard Key Derivation Identity-Based Encryption (WKD-IBE) [1] as a suitable policy-aware encryption scheme to implement RDE for the RTree policy type. Typically, IBE [25] (or an IBE variant including WKD-IBE) is instantiated with a single centralized Private Key Generator (PKG) that issues private keys to all participants. This does not meet the goals of WAVE, because the PKG is a central trusted party. In RDE, however, our insight is to *instantiate a WKD-IBE system for every entity— so there is no central PKG*.

A WKD-IBE system consists of a master secret and public key pair $(\text{WKD-IBE.}\mathsf{msk}, \text{WKD-IBE.}\mathsf{mpk})$. A message m is encrypted using the master public key WKD-IBE.mpk and a fixed-length vector of strings, called an ID: $\text{WKD-IBE.Enc}(\text{WKD-IBE.}\mathsf{mpk}, \mathsf{ID}; m)$. Using msk, one can generate a secret key for a set of IDs. This set is expressed as an ID with some components replaced by wildcards, denoted ID^*. The secret key $\mathsf{sk}_{\mathsf{ID}^*}$ can decrypt an encrypted message, $\text{WKD-IBE.Enc}(\text{WKD-IBE.}\mathsf{mpk}, \mathsf{ID}; m)$, if ID^* and ID match in all non-wildcard components.

The key thing to note here, is that given WKD-IBE.mpk, an entity can encrypt a message under any ID they choose, such that a specific $\mathsf{sk}_{\mathsf{ID}^*}$ is needed to decrypt it. This encryption process happens without any communication with the holder of WKD-IBE.msk. Specifically, a message can be encrypted under an ID before the corresponding secret key has even been generated. This means we can avoid the communication problem presented by the standard public key encryption approach.

In RDE, every policy p has an associated WKD-IBE ID called a *partition*. The partition corresponding to policy p is denoted $P(p)$. P is essentially a mapping function from a complex object with resources, permissions and expiry (p) to a vector of strings (ID). When issuing an attestation A, an entity encrypts it using $P(A.\mathsf{policy})$, in the WKD-IBE system of $A.\mathsf{subject}$: $\text{WKD-IBE.Enc}(\text{WKD-IBE.}\mathsf{mpk}_{A.\mathsf{subject}}, P(A.\mathsf{policy}); A)$.

Furthermore, the issuing entity generates secret keys in its own WKD-IBE system, suitable to decrypt messages encrypted under $P(A.\mathsf{policy})$, and includes them in the attestation. Let $Q(A.\mathsf{policy}) = \{\mathsf{ID}^*{}_i\}_i$ represent the set of IDs suitable for decrypting attestations encrypted under $P(A.\mathsf{policy})$, then A includes $W = \{\text{WKD-IBE.KeyGen}(\text{WKD-IBE.}\mathsf{msk}_{\mathsf{Issuer}}; \mathsf{ID}^*{}_i)\}_{\mathsf{ID}^*{}_i \in Q(A.\mathsf{policy})}$.

Due to the complex nature of a policy object, especially expiry, the functions P and Q which are used to determine the encryption ID and decryption IDs from a policy are not straightforward.

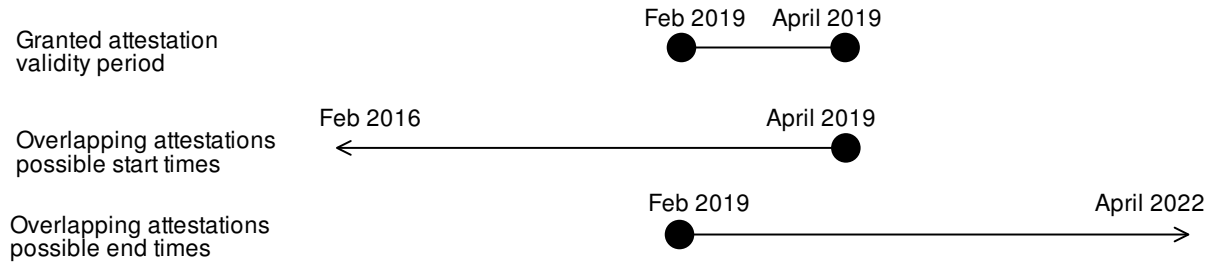


Figure 5.2: The possible ranges of time that the start and end time of an overlapping attestation could lie in

We refer to this pair of functions as the *partition map*. Below we develop the partition map for the RTree policy type.

Partition map for RTree.

To define P, consider that an RTree policy p_1 consists of a resource prefix as defined in §3.3 (matching multiple resources) and a time range during which the permission is valid (the validity range). A different RTree policy p_2 is considered compatible with p_1 if the resource prefix matches and the validity ranges overlap. To express the start and end of the validity range as a WKD-IBE ID, we define a time-partitioning tree of depth k over the entire supported time range; now any time in the supported time range can be represented as a *vector* representing a path in the tree from root to leaf. A WKD-IBE ID is a length-n vector: to represent attestations with a certain time range, we choose k of those n components to encode the valid-after time, and another k components to encode the valid-before time. The remaining $n - 2k$ components are used for the resource prefix. For example, an attestation policy that is valid from January 1st, 2019 to March 31st 2019 on a resource prefix of `namespace/building/*` might be encoded into a 14-component WKD-IBE ID as: `namespace,building,*,*,*,*,*,*,2019,January,1,2019,March,31`.

When granting an attestation for an RTree policy, the issuer encrypts the attestation contents under the resulting WKD-IBE ID $= P(A.\text{policy})$. Note that for a time tree of depth k, and a resource prefix of length ℓ, WKD-IBE must be instantiated with at least $n = 2k + \ell$ components.

The issuer must also include the policy-specific WKD-IBE keys from their own system in the attestations, generated with ID*s $Q(A.\text{policy})$, so that upstream attestations with compatible policies can be discovered. We define Q for RTree as: let E be a set of subtrees, each represented as a *prefix* of a time vector (i.e., a vector where unused components are wildcards), that covers the time range from the earliest possible encryption start time to the end of the time range of the attestation's validity. Let S be a set of subtrees that covers the time range from the start of the attestation's time range to the latest possible encryption time. Attestations in this scheme have a maximum validity of three years so this limits how long the start and end ranges need to be. Q returns ID*s corresponding to the Cartesian product $S \times E$ with each ID* also containing the policy's resource prefix. This allows any upstream attestation with an overlapping time range and compatible resource prefix to be decrypted by one of the secret keys in this attestation.

The decryption key id generation (Q) is best understood from example. Consider Fig. 5.2.

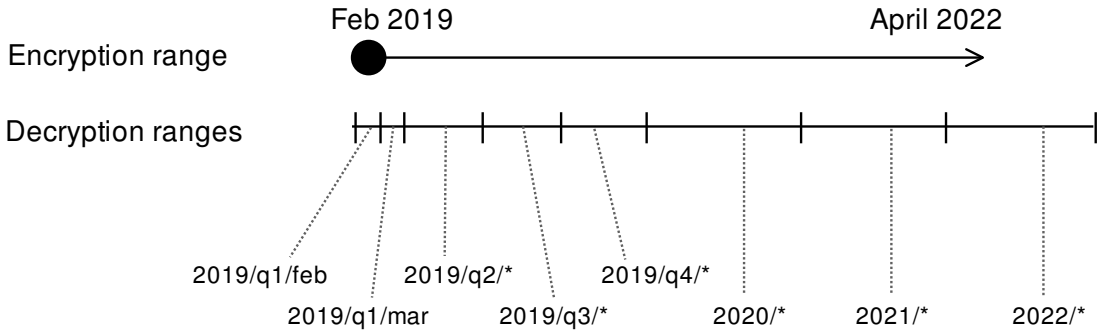


Figure 5.3: The key ID*s that Q would generate for the end time of the example attestation

At the top is an example attestation A granted in February 2019 with an expiry in April 2019. For the sake of this example, expiries have a granularity of a month and we use a $k = 3$ time partitioning tree of year,quarter,month. The maximum validity period for an attestation is 3 years, so an attestation B whose validity range overlaps with A's must have a start time that is somewhere between Feb 2016 (3 years before the start time of A) and April 2019 (the end time of A). If the start lay before Feb 2016, then the end would have to lie before Feb 2019 due to the three year limit, so it would not overlap. If the start lay after April 2019 then it would also not be overlapping. Similarly, there is a limited range that B's end time could fall in. If it lay before Feb 2019 then it would not be overlapping and if it lay after April 2022 then the start time would be after April 2019 so it would not be overlapping.

Let us just consider the end time. As can be seen in Fig. 5.3, Q would generate a set of ID*s such that any end time in the allowed range would correspond to one of the ID*s. We generate ID*s that go a bit past the possible end time (and a bit before in the possible start time case) because it doesn't really matter (decryption will be prevented by other portions of the complete ID*) and it allows us to use coarser-grained keys leading to fewer generated keys overall.

The process is similar for the start range, leading to 5 generated ID*s for 2016 to Feb 2019. The final ID*s generated by Q is then the concatenation of the resource prefix onto the Cartesian product of the set of start range ID*s and the set of end range ID*s leaving to a total of $7 \cdot 5 = 35$ keys that must be generated to allow for the decryption of any overlapping attestation. With the week-level granularity used in WAVE, the total number of keys is usually near 120, but varies depending on exactly how many keys need to be generated to represent a given start/end range.

The use of WKD-IBE leads to this Cartesian product where multiple "attributes" appear in parallel. This can be avoided by using KP-ABE [70], where the size of the key material would be proportional to the sum of these different attributes rather than the Cartesian product, but we have not investigated this due to the belief that the attribute based encryption would be slower for decryption. If the size of attestations becomes more important than the performance of decryption operations, it would be worth investigating KP-ABE for RDE instead of WKD-IBE.

5.4 Efficient Discoverability

In the scheme above, attestations are encrypted under the partition in the subject's WKD-IBE system. Unfortunately, it is subject to two shortcomings. First, a WKD-IBE ciphertext hides the message that was encrypted, but not the ID used to encrypt it; an attacker who guesses the ID of a ciphertext can efficiently verify that guess. Thus, every encrypted attestation leaks its partition. The second and more serious problem is that attestations are not efficiently discoverable. To understand this, suppose that Bob has issued many attestations $A_1, \ldots, A_n$ for Alice, with different policies. After this, an attestation B is granted to Bob. Alice might be able to form a proof using B and one of the A_i, but she does not know which of the A_i has a policy that intersects with $B.\mathsf{policy}$. As a result, she does not know which private key to use to decrypt B, and has to try *all* of the private keys conveyed by the A_i. This is infeasible if n is large, and becomes a vector for denial of service attacks.

If Alice knows B's partition, then the problem is solved—Alice can locally index the private keys she has from Bob's system, and efficiently look up a key that can decrypt B. However, B cannot include its own partition in plaintext, because it may leak part of $B.\mathsf{policy}$.

We solve this by encrypting the partition and storing it in the attestation along side the main encrypted body. For this outer layer of encryption we use a more standard identity based encryption (denoted IBE) that does not permit extracting the identity from the ciphertext [87, 83] because we do not need wildcards. As with the WKD-IBE scheme, every entity has its own system, removing the centralized PKG. The ID used to encrypt the partition is called the *partition label*, and is denoted $L(A.\mathsf{policy})$. For the RTree policy type, it is the RTree namespace of $A.\mathsf{policy}$. We expect users to have far fewer unique keys for this outer layer, so they can feasibly try all the keys they have.

We also move the WKD-IBE ciphertext under this IBE encryption so that the partition cannot be extracted (unlike the IBE scheme, the WKD-IBE ciphertext does not hide the ID). Finally, we include an IBE key from the issuer's IBE system, to allow the subject to discover the partition of upstream attestations. Because the partition label is simpler in structure than the partition, a single IBE secret key whose ID is $L(A.\mathsf{policy})$ is sufficient. So far, what gets stored in the attestation is:

$$\begin{aligned}&\text{IBE.Enc}(\text{IBE.mpk}_{A.\mathsf{subject}}, L(A.\mathsf{policy}); P(A.\mathsf{policy})||\\ &\text{WKD-IBE.Enc}(\text{WKD-IBE.mpk}_{A.\mathsf{subject}}, P(A.\mathsf{policy}); W||I))\end{aligned} \tag{5.1}$$

where W is defined as above, and

$$I = \text{IBE.KeyGen}(\text{IBE.msk}_{\mathsf{Issuer}}; L(A.\mathsf{policy}))$$

denotes the IBE secret key from the issuer's system.

Security Guarantees

We explain here at a high level how the policy-aware RDE restricts the visibility of attestations when used with RTree. Formal guarantees are given in Appendix A.

In summary, for each attestation A granting permission on a namespace: entities who have not been granted permissions in that namespace in a path from $A.\mathsf{subject}$ can only see the subject and

revocation commitment. Entities who have been granted some permissions in the namespace in a path from $A.\mathsf{subject}$ can see the partition (in essence what key is required to decrypt it). An entity e can decrypt an attestation A and use it in a proof if there exists a path, from $A.\mathsf{subject}$ to e where adjacent attestations (including A) have intersecting partitions. Issuers can encrypt under IDs before the corresponding private keys exist, so we introduce no ordering requirements and no interactivity requirements.

Thus, even though policy-aware RDE permits some entities to see more attestations than strictly needed to create a proof of authorization, it still provides a significant reduction in visibility when compared to structural security. We formalize the security guarantees of RDE in Appendix A.

A number of potential side channels are out of scope for WAVE, and can be addressed via complementary methods. Our storage layer does not provide any additional confidentiality, so compromised storage servers can see the time of each operation (e.g., when encrypted attestations are stored), which encrypted attestations are fetched, as well as networking information of the packets arriving at the storage servers (which could be protected via Tor [41], a proxy, or other anonymous/secure messaging methods [36]).

Revocation.

Although revoked attestations cannot be used in a proof due to the commitment revocation scheme described in §3.5, they still confer the ability to decrypt upstream attestations. Therefore we consider them part of the graph in the formal guarantees (Appendix A).

This can be mitigated by keeping expiry times short and reissuing the attestations. As there are no ordering or interactivity requirements, short expiries are easy to implement. For example, if attestation `#1` in Fig. 5.1 were to expire and be reissued, it would not require the reissue of any other attestation.

Integrity.

Finally, to maintain integrity, the issuer signs the attestation with a single-use ephemeral key $(\mathsf{pk}_e, \mathsf{sk}_e)$: $s_1 = \text{Sign}(\mathsf{sk}_e; A \setminus s_1)$, where $A \setminus s_1$ denotes the entire attestation except for s_1. Then, the issuer includes s_1 in the attestation in plaintext. The use of an ephemeral key ensures the signature does not reveal the issuer's public key. The issuer includes the outer signature in the plaintext header of the attestation. The issuer signs the ephemeral key pk_e with their entity private key, $s_2 = \text{Sign}(\mathsf{sk}_{\mathsf{Issuer}}; \mathsf{pk}_e))$, creating a short signature chain that ensures the attestation cannot be modified or forged. The issuer includes s_2 in the attestation *encrypted*, to avoid revealing the issuer's public key. In forming a proof, the verifier is allowed to decrypt s_2, allowing the verifier to verify s_2 and then s_1.

5.5 Reducing Leakage in Proofs

The methods discussed above ensure that a prover is able to decrypt all the attestations that it requires to build a proof. However, if a participant simply assembles a list of decrypted attestations into a proof and gives those attestations to a verifier, the verifier learns not only the attestations in that proof, but also the WKD-IBE keys in those attestations, which it can use to decrypt other attestations not in the proof. To solve this, we split the attestation information into two *compart-*

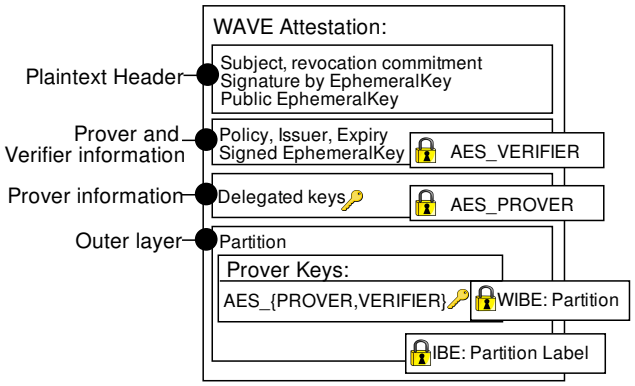


Figure 5.4: WAVE private attestation structure. The locks indicate the key used to encrypt the content.

ments, one for the prover (that includes keys it must learn for decrypting other attestations) and one for both the prover and the verifier (that includes the policy, the issuer, expiry etc). We encrypt the prover compartment with k_{prover} and the prover/verifier compartment with k_{verifier}, both symmetric keys freshly sampled for each attestation. k_{prover} and k_{verifier} are encrypted using WKD-IBE. This allows the prover to reveal to the verifier the necessary parts of an attestation by sending it the AES verifier key k_{verifier}, without allowing the verifier to decrypt other attestations.

The final structure of the attestation is shown in Fig. 5.4. The complete attestation contains (in addition to the plaintext header):

$$\text{Enc}(k_{\text{verifier}}; A.\mathsf{policy}||s_2) \qquad \text{(Prover/Verifier Section)}$$

$$\text{Enc}(k_{\text{prover}}; W||I) \qquad \text{(Prover Section)}$$

$$\begin{aligned}&\text{IBE.Enc}(\mathsf{mpk}_{A.\mathsf{subject}}, L(A.\mathsf{policy}); P(A.\mathsf{policy})||\\ &\text{WKD-IBE.Enc}(\mathsf{mpk}_{A.\mathsf{subject}}, P(A.\mathsf{policy}); k_{\text{prover}}||k_{\text{verifier}}))\end{aligned} \qquad \text{(Outer Layer)}$$

5.6 Discovering an Attestation

Each user's WAVE client maintains a *perspective subgraph* with respect to the user's entity, which is the portion of the global authorization graph visible to it. For each vertex (entity) in the perspective subgraph, the client "listens" for new attestations whose subject is that vertex (entity), using the Get and IterQueue API calls to the storage layer. For every attestation A received, the WAVE client does the following:

1. The client adds edge A to the perspective subgraph.

2. The client searches its local index for IBE keys received via attestations from A.subject, and tries to decrypt A's outer layer using each key. If none of the keys work, it marks A as **interesting** and stops processing it.
3. Having decrypted the outer layer in the previous step, the client can see A.partition. It searches its index for a WKD-IBE key received via attestations from A.subject that are at least as general as A.partition. Unlike the previous step, this lookup is indexed. If the client does not have a suitable key, it marks A as **partition-known** and stops processing A.
4. Having completed the previous step, the client marks A as **useful** and can now see all fields in A. The client adds WKD-IBE and IBE keys delegated via A to its index, as keys in the systems of A.issuer.
5. If the vertex A.issuer is not part of the perspective subgraph, then the client adds it and requests the storage layer for all attestations whose subject is A.issuer. They are processed by recursively invoking this algorithm, starting at Step 1 above.
6. If A.issuer is already in the perspective subgraph:
   - For each IBE key included in A, the client searches its local index for **interesting** attestations whose subject is A.issuer, and processes them starting at Step 2 above.
   - For each WKD-IBE key, the client searches its local index for matching **partition-known** attestations whose subject is A.issuer, and processes them starting at Step 3.

This constitutes a depth-first traversal to discover newly visible parts of the authorization graph revealed by A.

5.7 Extensions

Our RDE construction for RTree is performant but allows an entity to see attestations not required for correctness (i.e. partition-compatible attestations that are not usable in a proof, as defined in Appendix A). One reason for this is the use of a resource prefix as opposed to the full resource. If the full resource were used, then we would have a problem with sequences of attestations that are "narrowing". For example if A grants `a/*` to B and then B grants `a/b/c` to C. C could use those two attestations in a proof but the keys in the $B \rightarrow C$ attestation are not capable of decrypting the `a/*` attestation because it is more general (less qualified) than the `a/b/c` key that C obtained from B. With a resource prefix, although the full URI differs, both of these attestations would be encrypted with `a/` as the resource prefix so there are no correctness problems.

The resource prefix is a clean solution to the problem because it lets the end users reason about the visibility of their attestations without understanding the cryptography: if the resource prefix encompasses the department, then every attestation is potentially visible to people having permissions to resources within that department. There is an alternative solution, however, which is to include keys capable of decrypting less qualified ID^*s. The current number of keys that need to be included is $p \cdot s \cdot e$ where p is the number of resource prefix keys (always 1), s is the number of keys encoding the validity start and e is the number of keys encoding the validity end. If we included keys for decrypting less qualified ID^*s then p would encode the become equal to the length of the resource URI as we would need a key for each level of qualification (i.e. `a/b/c`,

Operation	AMD64	ARMv8
Create attestation	43.7	445
Create entity	8.9	88.5
Decrypt attestation as verifier	0.48	4.44
Decrypt attestation as subject	3.87	44.0
Decrypt delegated attestation	6.22	67.9

Table 5.1: Object operation times [ms].

`a/b*`, `a/*` and $\star$ in the example above).

As mentioned before, the cross product can be avoided by using KP-ABE [70], which becomes especially compelling in this scenario as the key material would be linear in the number of different options, rather than having this multiplicative relationship.

The advantage of using the full resource prefix is that the number of attestations that are decryptable but not required for correctness is lower. The disadvantage is that the visibility of an attestation is no longer easy to reason about: there is no resource prefix that bounds the visibility. Visibility could be wider or narrower than you expect, depending on which attestations other entities have granted. For example if A grants `d/e/f` to B, B grants $\star$ to C, and C grants `a/b/c` to D then D is capable of decrypting the `d/e/f` attestation even though they cannot use it in a proof, which may be counter intuitive to C as it is a consequence of B granting $\star$ which was not in C's control. With a resource prefix, you can always assume a worst case of "everything having this prefix is visible" and know that if the prefix is not the same, there is no scenario that would cause the attestation to become visible. For this reason, we have stayed with the resource prefix model, although there are benefits to both.

5.8 Privacy Micro Benchmarks

The performance of WAVE is dominated by two main factors: the cost of the core cryptographic operations required to perform RDE, and the cost of storing, indexing and retrieving the private keys required to decrypt RDE-encrypted attestations. The indexing cost is more of a consequence of the system implementation rather than RDE itself, so we relegate the evaluation of that to §6.10. Here we characterize the cost of the cryptography that RDE uses by benchmarking each operation in isolation.

Table 5.1 shows the operation times for the basic WAVE 3 operation. These are the times measured by a client using the External API, but without message serialization and deserialization times. The measurements are performed on an Intel i7-8650U AMD64 CPU with 4 cores (8 with hyperthreading), representative of a standard modern laptop. Additionally they are performed on a Raspberry Pi 3, indicative of a low-cost IoT-class ARMv8 platform. Both platforms are using 64-bit math.

The **create attestation** operation is predominantly the time taken to generate W which are the decryption keys corresponding to $Q(A.\mathsf{policy})$. Once Q yields the IDs, the keys are generated

across multiple cores. This is the only operation in the table that uses more than one core.

The **create entity** operation is the generation of the keypairs present in an entity, namely the IBE and WKD-IBE keys used for RDE, the Ed25519 key used for signing and the Curve25519 key used for standard public key encryption. This time does not include the time taken to generate a symmetric key using PBKDF2 if a passphrase is required. The passphrase key derivation is tuned to take 500 microseconds so as to not significantly slow down normal use of entities while still providing a barrier to brute forcing of entity passphrases.

The **decrypt attestation as verifier** operation is the time taken to decrypt the verifier compartment of an attestation given the AES key k_{verifier}. This time contributes to the time taken to verify a proof, so it is essential that it is efficient.

The **decrypt attestation as subject** operation is the time taken to decrypt both the verifier and prover compartments when the decrypting entity is the subject of the attestation. This is slightly faster than decrypting a delegated attestation because the prover compartment can be decrypted using the Curve25519 key of the subject instead of the IBE and WKD-IBE keys. This is achieved by encrypting the symmetric keys that would typically be obtained from the WKD-IBE decryption operation (AES PROVER and VERIFIER in Fig. 5.4) under the Curve25519 key of the subject and including this additional ciphertext along-side the IBE ciphertext.

The **decrypt delegated attestation** operation is time taken to decrypt an attestation where the full RDE protocol must be used. It requires decrypting the IBE ciphertext to obtain the WKD-IBE ciphertext, which is then decrypted to obtain the AES PROVER and VERIFIER keys. This is the common case when building the perspective graph.

Although the cryptography used here is more complex than, say, the RSA signature in a JSON Web Token found in OAuth2, it is still fast enough to not significantly impact the latency of application-level operations. In particular, creating an entity or attestation is typically done in response to a human typing a command or clicking a button in a user interface. Any delay of less than 50ms is imperceptible in a user interface context [12, 29], so the creation operations are fast enough. We show that the decryption operations are fast enough in §6.10.

5.9 Generalization to Other Policy Types

The RDE technique generalizes beyond the RTree policy described above. This section provides a more formal and general description of WAVE's mechanism for encrypting and decrypting attestations, agnostic to the policy type.

Definitions and Notation

We begin by establishing definitions and notation.

Notation 1 (ID)**.** *With respect to IBE (identity-based encryption) or an IBE variant, we use* ID *to denote an ID that can be used for encryption, and* ID^* *to denote an ID that can be used for key generation.*

The distinction between ID and ID* is important for expressive IBE variants, as the two types of IDs may have different forms. In WKD-IBE for example, an ID* may contain wildcards in some components, whereas ID may not. Similarly, for key-policy attribute-based encryption schemes, ID corresponds to a set of attributes, and ID* corresponds to an access tree.

Definition 1 (Policy Type)**.** *A **policy type** C is a tuple $C = (E, P, Q, F, L, M)$.*
$E = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$ must be an IBE variant (or ABE variant) in that a message can be encrypted using an ID, without any public-key generation process. The same applies to $F = (\mathsf{Setup}, \mathsf{KeyGen}, \mathsf{Enc}, \mathsf{Dec})$, with the additional stipulation that F must be anonymous*, i.e., ciphertexts in F must hide both the message contents and the* ID *used to encrypt the message. P is a map from policies to* ID*s in E, and L is a map from policies to* ID*s in F. Q is a map from policy to a* set *of* ID**s in E, and M is a map from policy to a set of* ID**s in F.*

Definition 2 (Partition)**.** *Let C be a policy type $C = (E, P, Q, F, L, M)$. A **partition** is an* ID *in the encryption scheme E. The partition of an attestation A with respect to C, denoted as $A.\mathsf{partition}$ (C is understood from context), is defined as $A.\mathsf{partition} = P(A.\mathsf{policy})$.*

Definition 3 (Partition Label)**.** *Let C be a policy type $C = (E, P, Q, F, L, M)$. A **partition label** is an ID in the encryption scheme F. The partition label of an attestation A with respect to C, denoted as $A.\mathsf{partition\_label}$ (C is understood from context), is defined as $A.\mathsf{partition\_label} = L(A.\mathsf{policy})$.*

Definition 4 (Precedes/Follows/Adjacent)**.** *Let A and B be two attestations in the same WAVE system. We write $A \rightarrow B$ and say "B **follows** A" (or equivalently, we write "$B \leftarrow A$" and say "A **precedes** B") if $A.\mathsf{subject} = B.\mathsf{issuer}$. A and B are **adjacent** if $A \rightarrow B$ or $B \rightarrow A$.*

Definition 5 (Path)**.** *Let x and y be entities. $(A_1, \ldots, A_n)$ is a **path** from x to y if either $n > 0$ and $A_1.\mathsf{issuer} = x$, $A_n.\mathsf{subject} = y$, and $A_i.\mathsf{subject} = A_{i+1}.\mathsf{issuer}$ for all $i \in \{1, \ldots n-1\}$, or $n = 0$ and $x = y$.*

For each policy type $C = (E, P, Q, F, L, M)$ supported, each WAVE entity e instantiates E-type and F-type cryptosystem. The public parameters, needed for encryption, are publicly available and are denoted $e.\mathsf{pp_E}$ and $e.\mathsf{pp_F}$. The master keys, needed for key generation, are stored privately on the WAVE client that generated e and are denoted $e.\mathsf{msk_E}$ and $e.\mathsf{msk_F}$.

5.10 Anonymous Proof Of Authorization

In addition to encrypting the attestations, the WKD-IBE keys can be used for anonymous proofs. Anonymous proofs offer a way for entities to give a coarse grained proof of authorization without revealing their identity or how they obtained the permissions, which is especially useful if the verifier is not a trusted party. The proof is described as coarse grained, because the anonymous proof does not specify an exact policy the way a standard proof does, but rather a partition $P(policy)$. As the mapping function P discards some information from the policy, there could be numerous policies corresponding to a given partition. Nevertheless, an anonymous proof remains a useful

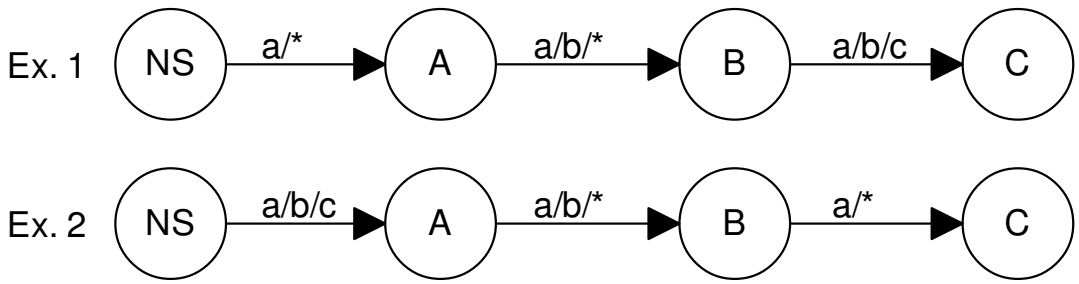


Figure 5.5: Two examples of delegation. Ex 1 is *narrowing* and Ex 2 is *broadening*.

tool where the fine-grained nature of a full proof is not critical. As an example, in Chapter 8 we describe WAVEMQ which uses anonymous proofs so that message routers can opportunistically discard messages from clients that definitely do not have permissions. It is acceptable if some messages slip through, however, because the final message recipient verifies the full proof.

To explain how anonymous proofs are formed, it is worth describing a simple (but insecure) baseline protocol. Consider Fig. 5.5, Example 1. In this example, the resource prefix used in P is the first element of the URI, so `a/*`. C has received permissions on `a/b/c` in namespace `ns`. When performing the RDE discovery process, C also learned the W keys included in the first `a/*` attestation granted from the namespace. While these keys are not used for attestation decryption (it is not meaningful for an entity to grant permissions to the namespace), possession of these keys means that C must have decrypted a path where each attestation's policy was compatible with the next one. The baseline anonymous proof for a policy p in namespace NS is, therefore, to yield a key from $W = Q(p)$ in the NS system. Only the key corresponding to the current moment in time needs to be shown, not the whole set of W. By showing this key, the prover shows the verifier that there must exist a path of attestations granting permissions to the prover on a policy having a partition corresponding to that key. To verify the key, the verifier can encrypt a random nonce under the partition being proven (anyone can encrypt under any ID without having a specific key) and then attempt to decrypt the ciphertext using the key it received. If the decryption succeeds, the verifier knows the key is valid.

Of course, sharing this key k is insecure, because it lets the verifier use that key for decryption or further anonymous proofs. We solve this in the final scheme by generating a specific ephemeral key k_e from k that is not useful for decrypting anything and cannot be used for further anonymous proofs. To do this, let the WKD-IBE ID have a component that is not used for normal encryption. This does not interfere with the normal RDE use of the key because it is always left as a wildcard. Now, when wanting to transform k into k_e, the proving entity hashes the message that the proof would be attached to and generates k_e where the reserved slot now contains the message hash (every other component is the same as in k). This qualified key is now useless for decrypting other attestations or messages (because every other ciphertext would have the reserved slot left as a wildcard). The verifier will, as before, create a ciphertext corresponding to the ID, this time with the message hash appearing in the reserved slot. If k_e decrypts the ciphertext, then the verifier knows the key is valid. Furthermore, because it is impossible to generate k_e without having k, the

verifier knows that the prover possesses k without the verifier learning k.

This scheme has some limitations inherited from the difference in specificity between a policy p and the partition $P(p)$. In example 1 of Fig. 5.5, it is necessary that the resource prefix that P identifies is `a/*`. If the resource prefix were, for example, `a/b/c` then the keys that C learns from the attestation granting `a/b/c` would be insufficient to decrypt the attestation `a/b/*`. The resource prefix must be common to all attestations, which means that it usually coarse grained, identifying a building or project rather than specific devices or services. This means that k and therefore k_e proves that C has permissions `a/*` even though in reality it only has permissions `a/b/c`. In general, an anonymous proof is always more broad than a full proof, proving a superset of the permissions that really exist.

Depending on the structure of the delegation graph, this problem might not appear. Consider Example 2 of Fig. 5.5. As the first attestation granting `a/b/c` contains keys that NS knows will never be used for decrypting an attestation, it can use the full resource in the partition instead of a prefix, because there is no danger of preventing A from decrypting useful attestations having NS as the subject. In turn, this means that upon decrypting the chain, C only learns k corresponding to `a/b/c` and the anonymous proof proves permission on the same resources that the full proof does. Depending on the use case, it might be possible to structure delegations in this way and avoid the difference in granularity between the partition and the policy altogether.

The anonymous proof is succinct, fast to verify, and verifiable by any entity.

5.11 Multicast End To End Encryption

The attestation encryption mechanisms discussed above can be thought of more generally: an entity can encrypt some message (the attestation) in a way that only entities having a specific set of permissions can decrypt it. This pattern is useful outside of attestation encryption, particularly for application-level end-to-end encryption.

We would like to ensure that end-to-end encryption and attestations encryption are disjoint; it should not be possible to use end-to-end encryption keys to decrypt attestations nor for attestation decryption keys to decrypt messages. This prevents user error – it is very confusing if the grant of an unrelated permission on a resource happened to enable end-to-end decryption because of how Q chose the resource prefix, a process that is largely hidden from the user. The typical method for this would be to use an entirely separate WKD-IBE system per entity for end-to-end encryption but this increases the size of entity public objects. We can instead introduce an additional reserved component into the WKD-IBE ID that is set to `attestation` when encrypting attestations and set to `e2ee` when encrypting application level messages. As with the reserved slot added for anonymous proofs, this ensures that although e2ee and attestation encryption keys are generated from the same key (WKD-IBE.msk), they are not interchangeable.

Additionally, we need to introduce a special permission that can be granted that, when seen by the WAVE service, lets it know that the keys to be generated and included in the attestation (W) need to include e2ee keys as well. We name this permission `wave::e2ee`.

At this point, we have baseline end-to-end-encryption. Consider again Fig. 5.5. Recall that in this scenario, the resource prefix used in P and Q is `a/*`. Interpret the figure as granting the permission `wave::e2ee` on the resources annotated on each edge. This causes the `a/*` attestation in example 1 and the `a/b/c` attestation in example 2 to include the e2ee keys in the namespace system. Consequently, C discovers these keys while discovering the attestations. To encrypt under a resource, for example `a/b`, an entity will form the WKD-IBE ID using P^1 that behaves like the partition mapping function P described above, but sets the reserved slot to `e2ee`. It will then encrypt the message using this ID. C can then use the keys it discovered to decrypt that message.

We call this scheme baseline, because it can be improved. There are two problems present in the scheme. The first is that we have inherited the somewhat-transparent notion of a resource prefix, embedded in P^1 and Q^1, but instead of being a security optimization that is invisible to the user, we are now actively affecting the permissions that the user is trying to reason about. For example, if Alice grants Bob an attestation with some specific policy, she need not think too carefully about what the resource prefix is because it only affects which other attestations Bob can decrypt, not which attestations Bob can use in a proof. With end-to-end encryption, however, the exact construction of Q^1 will affect which application-level encrypted messages Bob can decrypt. To prevent any unexpected over-grants that would arise from having a transparent resource prefix in Q^1, we alter the WAVE client so that when it observes that the `wave::e2ee` permission is being granted, it uses different partition mapping functions Q^2 and P^2 that uses the entire resource in the WKD-IBE ID rather than just a prefix. This forces the user to reason about which attestations are granted and which attestations are visible to the subject, but it prevents a scenario where WAVE automatically granted more permissions to the subject than the user intended. To see the impact of the changes in Q^2, consider Fig. 5.5 again. In example 1, the first attestation would use the partition `a/*`, the second under `a/b/*` and the third under `a/b/c`. The keys W^2 generated according to Q^2 use the same partitions. Consequently, when C learns the keys from the `a/b/c` attestation, none of these keys are capable of decrypting the `a/b/*` attestation, so the attestation discovery process will fail. This is the desired behavior, because if C had successfully decrypted the `a/b/*` attestation, this would have constituted an escalation of privileges as it would possess e2ee keys for `a/b/*` which B never intended on granting to C.

In contrast, example 2 of Fig. 5.5 poses no problems. Each attestation is "broader" than the previous, so the final e2ee keys that C will acquire are those for `a/b/c` which is correct. This means that, especially for e2ee grants but also for anonymous proofs, the user must think carefully about the topology of the graph. *Narrowing* e2ee permission grants, where an attestation grants fewer permissions than upstream attestations, could cause the subject to be unable to decrypt the attestations it needs to obtain decryption keys. In general, when granting e2ee permissions, an entity will need to grant multiple e2ee attestations, each granting fine-grained permissions, rather than a single coarse-grained attestation.

With this first problem introduced, the second advancement over the baseline scheme presents itself: if grants happen to occur chronologically, then narrowing grants like that in example 1 can be solved using a different mechanism. While WAVE as a whole does not introduce ordering constraints, the e2ee grants opportunistically take advantage of any ordering that might occur to increase the flexibility the user has in performing narrowing delegations. In example 1, imagine

that the attestations occurred in chronological order from left to right. At the time that A performs the grant of `a/b/*` to B, it is both in possession of the e2ee key `a/*` and also aware that B will be unable to decrypt the `a/*` attestation. Instead of relying on RDE for B to acquire the e2ee key in the namespace system (which would fail), A can generate the `a/b/*` key in the namespace system from the `a/*` key that it possesses and include that in the attestation to C. Now, when C decrypts the `a/b/*` attestation, it obtains the e2ee keys in the namespace WKD-IBE system corresponding to `a/b/*` without having to decrypt any further attestations. This scheme only works if attestations are granted in order.

In summary, for e2ee encryption, WAVE supports three of the four types of topology patterns: in-order broadening, in-order narrowing and out-of-order broadening. It does not support out-of-order narrowing such as example 1 if attestations were granted from right to left. Although it only applies to limited topologies, WAVE's end to end encryption is a powerful tool because it allows content created with *self-enforcing authorization*. Consider an email encrypted with this technique sent to a mailing list where membership is managed by WAVE. None of the intermediate servers or other attackers can read the message. Upon reception, however, the legitimate members of the mailing list would be able to decrypt it with no further out of band communication: a form of one-to-many end-to-end encryption where the sender does not need to know the recipients.

5.12 Protecting Name Declarations

Name declarations, discussed more completely in §3.9, are objects that form a global graph, similar to attestations. Unlike attestations which expose the subject of the attestation and encrypt the rest, name declarations expose the issuer and encrypt the rest. This means that name declarations are discovered by walking the graph forwards from an entity, rather than backwards.

We encrypt name declarations using the e2ee mechanisms described above, rather than the attestation encryption mechanism. Name declarations are associated with a resource, which can be thought of as the "directory" within which the name is listed. A typical example might be `namespace/department` as one would find in a normal directory system like Active Directory. To be able to decrypt name declarations, one must receive e2ee decrypt permissions on the resource corresponding to the directory the name declaration was placed in. To all intents and purposes, name declarations appear like application-level e2ee encrypted objects that are discovered using the notification mechanism that attestations use, described in §4.1.

5.13 Summary

Reverse Discoverable Encryption complements the ULDM storage layer, allowing us to meets the goals established in §2.1: it works with offline participants as it does not use communication between entities. It works with out of order grants as the encryption keys for policy-aware RDE can be generated before the corresponding decryption keys are generated. It protects permissions as only those parties that could possibly use an attestation in a proof are capable of decrypting it.

It does not introduce a central authority as we use IBE in a non-standard way that does not require a PKG.

In addition to encrypting attestations, the key dissemination function of RDE allow us to perform end-to-end encryption which, aside from application-level uses, also allows us to encrypt name declarations.

Chapter 6

System Design And Implementation

To validate the efficacy of graph based authorization as well as ensure that RDE functions as intended, we construct a full instantiation of our techniques. This chapter details the design and implementation of the WAVE daemon that performs authorization operations for applications and services. We begin with a high level system overview, followed by a justification for the architecture of that system and a discussion of the design choices made in each component.

6.1 WAVE 3 System Overview

When an application interacts with a traditional authentication/authorization system, such as LDAP, it typically uses a library that allows the application to interact at a high level such as "is this username/password correct?" or "what groups is this user in?". This is similar for OAuth where one would ask "is this token valid?" or "what scopes does this token grant?".

WAVE is engineered to provide a similar mode of operation and answer similar questions, whilst under-the-covers providing stronger security. For reasons further discussed in §6.2, WAVE is implemented as a daemon that typically runs on the devices that interact with WAVE. A service would interact with WAVE by using a lightweight library that uses loopback or IPC sockets to securely communicate with the daemon. By using this library, they can perform the operations described in prior chapters, such as building proofs, verifying proofs and encrypting messages.

Although services may be interacting over a socket, WAVE does not maintain a session as one might have with a database. Interactions are all independent, as one would have in a function call interface. This allows services that may be processing authorization operations for several different users to do so without fear of state getting confused between them. This is the same model used by other authorization libraries, so allows software to keep a familiar construction and mode of interaction when it is ported to use WAVE.

The WAVE daemon comes with a command line utility that permits users to perform WAVE operations interactively or in a script. This is used heavily for granting and revoking permissions as these are usually the result of a human action. Most of the time, services do not alter the authorization graph, and only require the generation and verification of proofs.

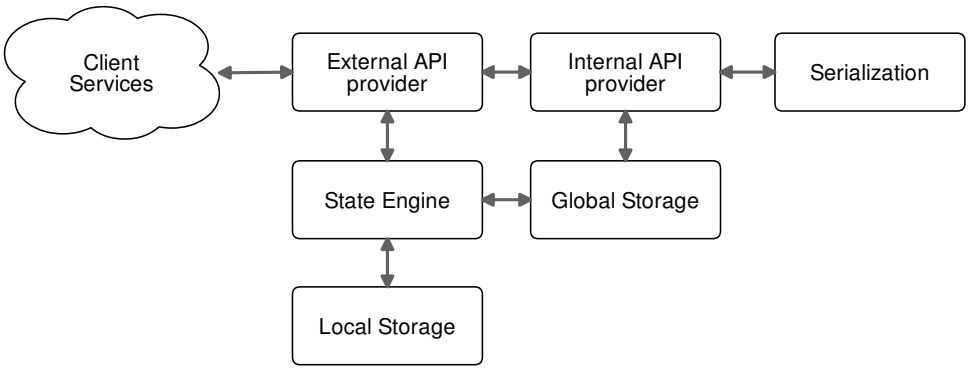


Figure 6.1: An overview of the components in the WAVE daemon.

WAVE is implemented in Go to ease cross-platform compilation. The source can be found in [8]. Certain parts of the cryptography, notably the IBE and WKD-IBE implementations from [78] are in standards-compliant C++. We have tested the implementation on ARMv7, ARMv8 (in 32 and 64 bit modes), i386 and AMD64 machines. WAVE is generally cross-platform compatible, although some optimizations in the compilation of the cryptography need to be turned off for older or virtualized CPUs.

The WAVE daemon is composed of clearly defined modules with interfaces between them, as shown in Fig. 6.1. Applications interface with the External API Provider through a GRPC interface. The external API functions are implemented as sequences of operations implemented in the Internal API provider and the State Engine. Those components in turn rely upon the Serialization module, the Global Storage module and the Local Storage module. Each of these is discussed below.

6.2 System Integration Pattern

The first design choice that influences the overall architecture of WAVE is how the application will interface with it. When a set of functionality is provided for applications to use, the most common integration pattern is that of a library. For example, if an application needs to read PNG images, it would link against the `libpng` library, importing the correct headers. In most cases, this linking is done dynamically, so that the shared object implementing the library can be upgraded independently of the application.

This approach is used by many security libraries, for example `libssl`, which is a very widely used library providing cryptographic primitives, such as AES and RSA encryption, as well as an implementation of TLS for secure channels of communication.

The initial implementation of WAVE was implemented as a library for use by python applications, but we quickly encountered difficulties with the library approach. The first is that the library needs to be re-implemented in several different languages in order to be a cleanly used from those languages. Some languages allow libraries written in C to be used (for example Go, Python, Java)

but this usually complicates the build process and applies restrictions on the resulting program. For example a Python or Java program using a C library cannot run on a different operating system from the one it was compiled on, whereas a pure Python or Java program is cross-platform.

The second issue is that many common languages are very inefficient at the math found in the cryptography that WAVE uses. For example a pure python implementation of the Ed25519 ECC library is an order of magnitude slower than the C implementation.

As WAVE became more complex, it became apparent that maintaining multiple parallel codebases in different languages would not be tractable for a research scale development.

The solution is to move towards a daemon architecture. WAVE versions 2 and 3 run as a daemon on the machine and services connect to the daemon using a simple *binding* library that connects over a loopback or UNIX domain socket. The API exposed by the daemon is engineered specifically to ease the development of bindings in multiple languages. In WAVE version 2, a simple text-based protocol was used to connect services to the daemon, making the bindings easy to write. In WAVE version 3 this principle is taken even further and the daemon offers a GRPC interface. GRPC is a widely used cross-platform cross-language RPC framework that will auto-generate the bindings for multiple languages. This drastically reduces the development effort spent on language bindings as the GRPC tools will auto-generate the vast majority of the required code.

In the daemon architecture, only a single implementation of the cryptographic operations needs to be written, so more effort can be invested in optimization. In the blockchain-based WAVE 2, the daemon has the advantage that only a single copy of the blockchain needs to be maintained across the system, and the daemon can keep this copy up to date even if the applications and services are only running periodically.

The advantages of dynamically linked libraries, such as reducing application footprints, providing independent upgrades, etc. also apply to the daemon architecture. In essence, the daemon acts as a dynamically linked library where the link is a network connection rather than a function call interface.

The daemon is often run in a different execution environment, for example at the host-level of a machine that is running several services in containers. In some cases, the daemon is run on a different physical machine to permit services running on embedded platforms such as Cortex-M0 to interact with WAVE primitives even though the cryptography would be very slow on those platforms.

The next few sections detail the construction of the WAVE daemon.

6.3 Internal API Provider

The Internal API Provider contains the core operations of WAVE, but does not interact with any state directly. When the External API invokes functions in the Internal API, it passes a context containing all the requisite state that is specific to the entity doing the operation. This entity is called the *perspective entity*. The stateless construction simplifies the structure of tests, as artificial state contexts can be passed to IAPI functions to test corner cases, rather than attempting to construct that state through a sequence of external API calls.

Entity Operations

New entities are created by invoking the `NewEntity` function in the Internal API (IAPI). This function will sample fresh keys for the Ed25519, Curve25519, IBE and WKD-IBE schemes. The public keys for these cryptographic schemes are assembled into a public entity object that is serialized using the serialization module. If requested to do so, the IAPI will also forward the created entity object to the global storage module to be published. The private entity object is, optionally, encrypted with a passphrase using a password based key derivation function (PBKDF2) to generate an AES-GCM key. The passphrase, if present, must be supplied to all future operations using the entity. This passphrase ensures that even if the file containing the private entity object is compromised, an attacker cannot use the file without knowing the passphrase. The use of PBKDF2 ensures that the attacker cannot brute force the passphrase used to encrypt the entity.

Private entity files and public serialized entities can be parsed by `ParseEntity` in the IAPI to yield an in-memory de-serialized form that can be passed to other functions, such as `CreateAttestation`. This function will verify that the entity is well formed and will convert all the cryptographic keys from their compact marshalled forms into the expanded form required to use them in cryptographic operations.

Attestation Operations

A new attestation can be created with the `CreateAttestation` function. This takes an attester entity secret, a public subject entity, a policy and an expiry. It implements RDE, so will generate RDE keys from the attester and assemble the various compartments of the attestation, described in §5.5. After serializing the compartments it will encrypt them using the RDE keys for the subject entity, following the protocol established in §5.4.

To implement the revocation scheme described in §3.5, a fresh secret is generated from the combination of the attesting entity's revocation seed and the ephemeral signing public key using a cryptographic hash function (SHA3-256). The hash of this secret is included in the attestation's plaintext compartment.

The final attestation object is serialized in canonical form using the serialization module and, if requested, is forwarded to the global storage module for publication. If published, the hash of the serialized form is also inserted into the appropriate notification queue to permit RDE discovery.

Name Operations

An entity can assign a human readable name to another entity by using `CreateNameDeclaration`. This takes an issuer private entity and a subject public entity, along with an expiry and the partition under which the name declaration will be encrypted.

The internal representation of the name declaration object is assembled and then passed to the serialization module to obtain the canonical ASN.1 encoded form. This serialized object is then encrypted using the partition and the namespace public entity. This encryption is done using the

end-to-end encryption scheme described in §5.11 rather than the attestation encryption scheme. If requested, the encrypted object is passed to the global storage provider for publication.

Proof Operations

The IAPI contains the logic for building proofs. The proof building routine is given a desired RTree policy and an instance of a State Engine that is specific to the proving entity. Using the state engine, the the proof builder can query for decrypted attestations (within the proving entity's perspective graph) that are granted from a specific entity. When building a proof that may contain multiple edges between two entities, a more complex algorithm (described in §6.9) must be used, but in the simple case where the proof consists of a single path from the namespace entity to the proving entity, the proof can be built by:

```
▷ policy is an RTree policy that we want to prove
▷ grantedFrom is a map from entity to a list of decrypted attestations they have granted
function BUILDPROOF(policy, grantedFrom)
    paths ← [i]∀i ∈ grantedFrom[policy.namespace] | i.policy ⊃ policy
    while paths ≠ ∅ do
        p ← head(paths)
        paths ← tail(paths)
        edges ← i∀i ∈ grantedFrom[p.subject] | i.policy ⊃ policy
        for e ∈ edges do
            if e.subject ∈ p then
                continue
            end if
            newpath ← p||e
            if e.subject = policy.subject then
                return newpath
            end if
            paths ← paths||newpath
        end for
    end while
    return ⊥
end function
```

This simplification of the proof building algorithm does not check that the path as a whole grants the policy, which is sometimes not the case even when the individual edges grant the policy as a result of the Time To Live (indirections) parameter limiting the re-delegation. Nevertheless, it serves as a simple example of the general approach towards building a proof. The actual approach used by WAVE is given in §6.9.

6.4 External API

The Internal API, provided by the Internal API Provider described above, is a function call interface API. The parameters and return values from the functions are in-memory representations that are not easily constructed by arbitrary applications as they are not easily serialized and deserialized. Further, the functions are stateless which puts the burden of maintaining the state required to complete each operation on the consumer of the Internal API.

To resolve this, the External API Provider refactors the Internal API such that the parameters and return values of all the functions are protocol buffer [66] structures, which can be automatically serialized and deserialized by GRPC. This also ensures that the structures can be represented in any language that supports protocol buffers. In addition to providing an alternate representation of the parameters that is more amenable to cross-application invocation, the External API also attempts to "humanize" parameters. For example, when creating an attestation, the client can pass the subject hash as a base64 encoded string instead of the normal raw byte array, and the EAPI will detect this and correct for it. This makes the API more forgiving, and especially improves the usability from languages like Python 2 where working with byte arrays is more clumsy than working with strings. Wherever possible, it also compensates for missing parameters. For example, a client can omit the location of an entity when granting it permissions and the EAPI will search for the correct storage server in decreasing order of likeliness. Omitting an expiry date causes the EAPI to fill in a "safe" default of 30 days.

In addition to making the parameters to operations more friendly, the EAPI also absolves the client from state management (other than storing the entity secret). IAPI functions all require the correct state context to be passed, but this state is very complex to maintain. The EAPI will construct the correct window into the entity-specific state maintained in the State Engine and pass it to the IAPI so that the client does not need to reason about which state is relevant or how to store it.

In general, an application just needs to store their entity secrets to pass as the perspective entity parameter to EAPI functions and all other state (such as the relevant portions of the decrypted persective graph or name declaration graph) will be filled in by the EAPI before passing the call to the IAPI.

6.5 State Engine

Although the canonical state store is the global ULDM storage, each entity must decrypt and index a portion of that state in order to be able to build proofs and resolve name declarations. The state engine is responsible for maintaining this decrypted/indexed state and making it available for the External API and, transitively, the Internal API to use.

State transitions are triggered by a *resync* event. This can either be a global resynchronization where the notification queue for every entity in the decrypted perspective graph is polled, or a local resynchronization where only specific entities are resynchronized. At the time of writing, only the global resynchronization is implemented as all the perspective graphs we have worked with

are small enough to synchronize rapidly. For entities with perspective graphs containing tens of thousands of entities, it would be useful to be able to synchronize portions of that graph on demand rather than the whole graph. This local resynchronization could be triggered, for example, by the expiry of an attestation where it is likely that a replacement attestation would involve the same entities.

As new attestations are discovered in the polled notification queues, the state engine attempts to decrypt them by following the procedure detailed in §5.6. As a quick summary of that procedure, the decryption process entails transitioning an attestation between four states: unknown, interesting, partition-known and useful. Unknown attestations are those that are not connected to the perspective graph in any way. All attestations begin as unknown with respect to a given entity before they begin graph discovery. When an attestation becomes connected to the perspective graph, i.e. the subject of the attestation appears in the perspective graph, it is deemed "interesting". Recall that there are two layers of encryption in an attestation (§5.4). If the attestation A is granted on a namespace N and there exists a path of decrypted attestations from $A.\mathsf{subject}$ to the prover where all of the attestations along the path are also granting permission on namespace N, then the outer layer can be decrypted and the attestation becomes "partition-known". If all the attestations along this path are additionally *compatible* (formally defined in Appendix A) then the inner layer of encryption can also be decrypted and the attestation becomes "useful".

The transition from partition-known to useful must be done carefully as there are concurrency concerns. Imagine we are simultaneously processing two attestations, A and B, and further that B contains the keys necessary to decrypt A. If we begin processing A before we fully decrypt B, but mark it as useful or partition-known after we finish processing B, then we will have a kind of deadlock where we will never re-attempt to decrypt A even though we acquired the keys in B. The solution is to ensure that the transition from partition-known to useful is atomic with respect to any other attestation's transition. In practice this means that we hold a mutex per perspective entity that ensures that an attestation is only added to storage once all the appropriate keys have been used to try and decrypt it and, conversely, a new key is only added to the storage once it has been tried against all pending attestations. This ensures the consistency of the perspective graph even if it is updating multiple entities at a time.

6.6 Indexed Local Storage

For each entity interacting with the WAVE daemon, there is a perspective graph containing all the attestations that have been decrypted by following the RDE discovery process detailed in §5.4 and summarized above. The local storage keeps that information in a persistent database, to facilitate both future RDE discovery and proof formation.

The state is partitioned, so that state belonging to one entity cannot be accessed by another entity. The operations supported by this storage are:

- Storing/Retrieving revocation checks, a persistent cache of the result of checking global storage for a commitment revocation with an adjustable expiry time. This reduces the number

of requests made to global storage when verifying proofs.

- Get/Update a list of entities that are *interesting*, i.e. whose notification queues will be inspected when updating the perspective graph.
- Get/Insert partition label keys, used for decrypting the outer compartment of attestations
- Get/Insert WKD-IBE keys, used for decrypting the inner compartment of attestations
- Move attestations and name declarations between the decryption states described in §5.4: interesting, partition-known, useful, revoked/expired.
- Query attestations and name declarations based on partition
- Resolving a name using decrypted name declarations
- Querying the structure of the perspective graph, such as attestations granted to or from a given entity.

The indexed local storage is built upon another layer of abstraction, the *low level storage* which is a key-value interface. This interface could be met by many existing databases such as LevelDB [65], RocksDB [51] or Badger [40]. We use LevelDB, although our experience in developing WAVEMQ (Chapter 8) would indicate that Badger might offer marginally higher performance.

Future work in this module may introduce an in-memory caching layer. The objects stored here are not particularly large but depending on the workload there are many requests for the same object. An in-memory cache could offer some performance improvements.

6.7 Global Storage

The Global Storage module abstracts the ULDM storage tier described in Chapter 4 so that the IAPI and State Engine can request objects from storage or place objects into storage without thinking about any of the cryptographic processes that go into validating the integrity of the storage.

Objects such as entities, attestations and name declarations have a *location* field that identifies the storage server they are located on. When referencing one of these objects by hash, the reference also includes this location field. This permits an attestation to be granted to a subject entity that resides on a different storage server from the issuing entity, for example.

In the configuration for the WAVE daemon, a list of storage locations along with symbolic names such as “default” can be given. This allows operations, such as creating an entity, to omit the storage server or refer to a storage server by a human readable name, and the daemon can resolve this to an actual server. The full reference to a server contains its domain name and port, along with the public keys of the server. This is used in addition to the TLS certificates to ensure that the remote endpoint is valid and it allows the signatures present on various ULDM messages to be validated.

The global storage module is responsible for performing the ULDM protocols described in Chapter 4. In summary it validates the proofs that are attached to responses from ULDM servers that show that the server is answering honestly. At this time, the WAVE implementation does not do anything special when it discovers that the ULDM server is dishonest, but one could imagine that it would forward the contradictory signed proofs to an auditor or other authority.

Storage operations are described in §4.1 and §4.2. We touch on implementation issues here:

PutObject places an entity, attestation, revocation or name declaration in a given storage server. The key is the hash of the object's ASN.1 serialization. This will additionally validate the merge promise returned from the ULDM server. For convenience, these functions take the in-memory representation objects and invoke the serialization functions transparently.

GetObject retrieves an entity, attestation, revocation or name declaration from a given storage server. This validates the inclusion proof (or non-inclusion proof if the object did not exist). If a merge promise is returned, the merge promise is validated. Depending on context, the caller of this function usually knows what the object is supposed to be, so there are utility functions for each variant (e.g. GetEntity) that automatically deserialize and validate the object before returning the in-memory representation.

Enqueue implements `NotifyAttestation` and `NotifyNameDecl` in §4.2. It places the hash of an attestation or name declaration at the end of a queue belonging to an entity, to allow for discovery later. If there are multiple clients attempting to enqueue at the same time, the ULDM server may return an "already-exists" error when a given queue entry is written to. There is a retry mechanism to resolve this problem, although generally attestations and name declarations are created fairly infrequently and the odds of collision are low.

IterQueue implements `GetNotificationAttestation` and `GetNotificationNameDecl` in §4.2. It retrieves a given queue entry or, after validating the non-inclusion proof, returns that the client has reached the end of the queue.

The global storage module is written to permit alternate storage backends (other than a ULDM). A blockchain based backend was planned, to prove that it is possible, but the implementation was never completed.

6.8 Serialization and Representation

The serialization module performs canonical encoding of WAVE objects from their in-memory representation to an exchangeable format.

There are some requirements placed upon the serialization of WAVE objects. The first is that the serialization must be deterministic: the hashes of objects are used to identify them, so if different versions of a serialization library are permitted to produce different results, there will be inconsistencies that will cause validation to fail. The second requirement is that the format must be efficient at encoding binary fields: there are many fields in WAVE objects that are large (100's of KB) binary objects, so it is essential that the serialization is succinct as possible.

These two requirements rule out most of the widely used serialization formats. JSON and XML the most widely used formats on the web, both are inefficient at encoding binary fields. Furthermore, JSON does not specify a strict ordering of fields and many implementations of JSON serialization will re-order the fields (this is true of Go's implementation). Msgpack offers efficient binary representation but different versions of msgpack will encode the same data using different msgpack primitives. Additionally msgpack implementations are free to re-order fields. Protocol buffers have strict field ordering but are free to change the exact binary representation of fields between versions or implementations.

The need for deterministic serialization for validation purposes is not new, this exists everywhere a data structure is signed. In some systems this is done by embedding serialized forms of an object as a field in the parent object. For example, in JSON this would be:

```
{
    "signature": "base64 encoded signature here",
    "message": "base64 encoded JSON here"
}
```

Instead of:

```
{
    "signature": "base64 encoded signature here",
    "message": {
        "from": "sender",
        "to": "recipient",
        "content": "hello world"
    }
}
```

This approach is used in Trillian [67], for example, where protobuf serialized forms of the map roots are embedded as byte array fields in messages, instead of the standard approach of having a sub-object.

Another approach, such as that used in TLS certificates, is to use a deterministic serialization format such as ASN.1 Distinguished Encoding Rules (DER). This means that an object encoding a given set of parameters will always yield the same binary representation, irrespective of which implementation or version of DER library is used.

Additionally, ASN.1 offers rich schema validation of objects, similar to protocol buffers and unlike JSON. This is useful as it shifts some of the burden of validating objects to the serialization library, rather than leaving all forms of sanity checking to the application.

WAVE uses ASN.1 DER encoding of all objects. Additionally, it heavily uses `EXTERNAL` types in ASN.1 which allows a field to be one of many types while retaining strict schema validation. For example, the signature field in a message can be one of multiple different signature types, identified by an Object IDentifier (OID). This permits WAVE to evolve over time, adding new and improved cryptographic schemes as and when they become available.

Throughout the course of WAVE's development, improvements to the cryptography have already happened once as the initial WKD-IBE implementation based on the BN-256 curve was discovered to be vulnerable to an attack reducing its security level to approximately 96 bits (from the expected 128 bits). This led us to develop a WKD-IBE implementation based on the BLS12-381 curve which is expected to have a security level of 128 bits. By clearly identifying the types of the keys, signatures and ciphertexts embedded in the various WAVE objects, the WAVE client can distinguish between the old and new curves cleanly.

6.9 Proof Building Optimization

Proof building is done in the Internal API. We gave a simple form of the algorithm in the IAPI section above, but that doesn't capture the full complexity of the proof building in WAVE. The aspect that makes the problem more interesting is that a proof can be a subgraph, instead of just a path. Consider the proof in Fig. 6.2. This proof grants both access to `ns/a/*` and `ns/b/*`.

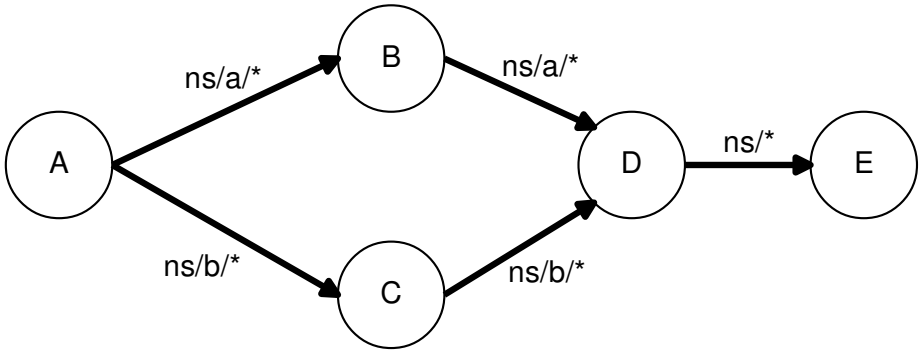


Figure 6.2: A proof consisting of a subgraph, rather than a single path

This structure of graph is common when you have a service that has been granted permissions by separate people. For example, a service may be doing reporting on inventory projections and require both customer purchase data and current inventory data. The customer purchase data may be managed by a different entity to the inventory data so you would end up with a graph like Fig. 6.2 where two different entities (B and C) grant permission to the same entity (D). In fact it is even common for a *single* entity to grant multiple attestations to the same entity. The permissions that D requires may not be known all at once and may evolve over time. It is natural to grant new attestations that just contain the missing permissions rather than revoking everything that exists and re-granting it all in one attestation.

While of course, a client could simply provide two independent proofs showing the paths ABDE and ACDE to prove the two authorizations independently, it is convenient to be able to prove multiple permissions with a single proof as it reduces the size of the proofs that need to be attached to messages. You could imagine the scenario where a group entity is granted multiple

permissions with different attestations and we would like member entities to be able to use the permissions of that group with a single proof rather than having to attach multiple proofs.

This functionality complicates proof building, as one can no longer prune edges that do not implement the full policy when performing the breadth first search as they may need to be combined with other edges in order to realize the policy.

The naive approach would be to compile a list of every path granting nonzero permissions between the namespace entity and the proving entity and then search for a subset of those paths that grant the required permissions. Unfortunately, depending on the structure of the graph, a full enumeration of all possible paths could be computationally expensive. Consider a graph where there are $N + 1$ entities with K attestations between each successive entity. The total number of paths here would be K^N and naively deciding on which subset of these paths to use in a proof would be $O(N2^{K^N})$ since there are K^N sets and evaluating each requires combining N policies. This would rapidly become infeasible to compute. An algorithm that chose the required subset at each link independently would run in $O(N2^K)$, which is substantially better, but is hard to generalize when there are multiple entities traversed in parallel as opposed to N sequential entities. It is, however, possible to think of the problem of choosing which attestations to combine together to reach a given policy as a form of the subset-sum problem. Although this problem is NP-complete, there are a few interesting algorithms that run in pseudo-polynomial time with a few caveats.

To rephrase the problem into a form more amenable to consideration as a subset sum problem, we need to change how the policies are represented. Recall that the essence of a policy is a set of permissions and a set of resource URIs the permissions are granted on. When we consider an attestation while building a proof, we are considering it in the context of a desired policy that we aim to prove. For each statement in the target policy, there is a set of permissions on a resource. We can expand these into a list of (permission, resource) tuples. We can assign each of these tuples an integer identifier that is a power of two, so that we can meaningfully work with sets of these tuples as integers, and the union of disjoint sets of tuples can be calculated with addition. The union of partially intersecting sets tuples can be calculated with bitwise AND.

Similarly, we need to process each attestation we encounter in the graph, converting its policy into a list of integers. We begin by forming the same list of (permission, resource) tuples from the statements in the attestation. We then process this list, assigning tuples the same integer identifier found in the target policy if the permission matches and the attestation statement resource is a superset of that in the target policy. We discard any tuple with a permission not found in the target policy or a resource that is a subset of that required by the matching permission in the target policy.

After performing this procedure, the permission graph looks as in Fig. 6.3. The edges can be represented by a single integer, represented in binary.

There are some additional complications that we need to factor into the formulation of the graph, however. Firstly, a policy has a *TTL* which denotes its maximum delegation depth. Consider the graph in Fig. 6.4. In this graph, if `E` were the prover, they could form a proof for the policy identified by `11`. If `F` were the prover, however, they can only form a proof for `01` as the $\overline{BD}$ attestation has a TTL of only 1 and can not be re-delegated by `E`.

The second complication is that we want to construct a proof that has the least number of edges in it, as this reduces the size of the proof that will be sent with each message. In a normal breadth

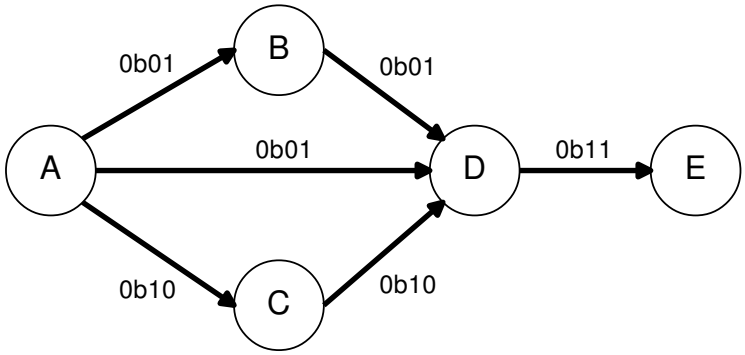


Figure 6.3: A graph where the policies on the edges have been replaced by bit sets

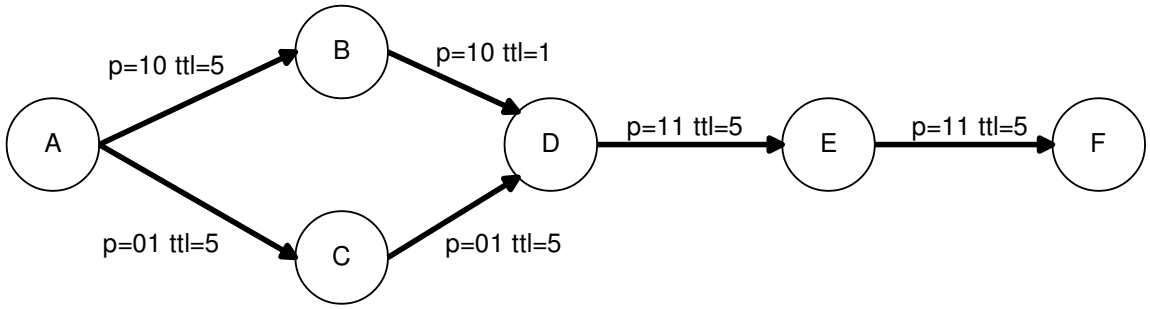


Figure 6.4: A graph where TTL comes into play

first search of a single path, the first path you find is guaranteed to be the shortest. This is not true when searching for a *subgraph* with a breadth first search. Consider Fig. 6.5. A breadth first search will first discover a proof along $A \rightarrow B \rightarrow C$. We call the total number of attestations in the proof its *weight* and in this case it is 4. The proof from $A \rightarrow C \rightarrow D \rightarrow E$ has a total weight of only 3, so it is a superior proof. In general, we must continue the breadth first search until we reach depth $W - 1$ where W is the weight of the lightest proof discovered so far. After $W - 1$ even a proof in the form of a singly-linked path would be heavier than the already-discovered proof, and we can terminate the search.

To summarize the goals of the proof building operation: we wish to discover the smallest subgraph (lightest weight proof) that grants a set of permissions and where the TTL of any given attestation is not exceeded. A proof is considered *complete* if it terminates at the proving entity, and is considered *intermediate* if it terminates at some other entity. Note that finding the lightest weight proof is ideal, but not required for correctness, so our algorithm need not be strictly optimal in that regard. Obtaining the correct permissions without exceeding any TTLs, however, is mandatory.

We refer to the TTL of a proof (as opposed to an attestation) as being the number of delegations that can be appended onto the end of the proof before one of its constituent attestations can no longer be applied. This is only really meaningful with respect to intermediate proofs, as a complete

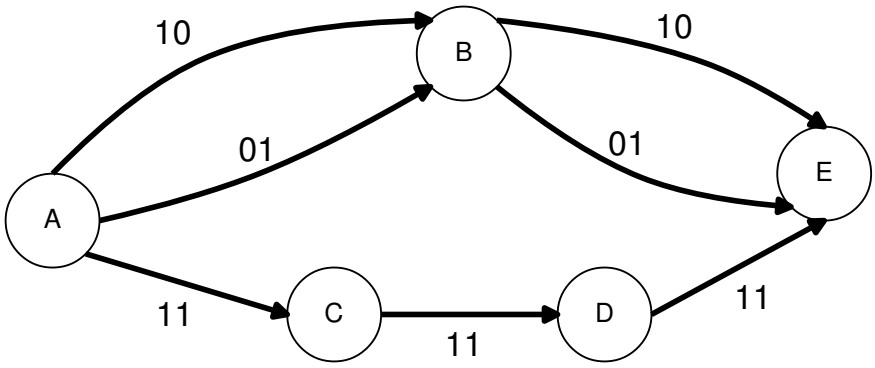


Figure 6.5: The first discovered path (ABE) is not the lightest weight

proof does not need any more delegations to be useful to the proving entity. With this, we can establish rules for comparing two intermediate proofs P and P' that lead from the namespace entity to an intermediate node. These rules will enable us to discard intermediate proofs along the way when we discover other proofs that are categorically superior, reducing the total complexity.

- P is superior or equal to P' if $P.weight <= P'.weight$ and $P.ttl >= P'.ttl$ and $(\sim P.bits)\&P'.bits = 0$. P' can be discarded.
- P is distinct from P' if $(\sim P.bits)\&P'.bits \neq 0$ and $(\sim P'.bits)\&P.bits \neq 0$ meaning neither set of permissions is a subset of the other. Both proofs must be kept.
- P is distinct from P' if ($P.weight < P'.weight$ and $P.ttl < P'.ttl$). Both proofs must be preserved as we do not know how many more delegations will be required until the proofs are complete, so we cannot know if we will need the higher TTL at the expense of the higher weight.

Our solution is as follows. Every node we traverse has a *Solution Table* mapping from a permission bitset to a list of "best" solutions for that bitset following the rules above.

We begin the breadth first search with the namespace authority node that has a solution table with one base-case entry: an empty intermediate proof that grants all the permission bits with the maximum TTL. We then populate the *border*, the queue of nodes to be traversed with the starting node. We care about the depth of our BFS, as this is our termination condition, so instead of the traditional single queue, we use a new queue for each depth, performing some checks before beginning to process the nodes in the next depth.

For each node in the border, we add it to the queue for the next depth and then update its solution table, which we discuss below. If that node was the proving entity, then we also update our maximum breadth first search depth to be one less than the weight of the lightest proof in the solution table. We continue iterating until there are no more nodes to traverse (border is empty) or until we reach that maximum depth.

The process discussed above is not that different from a normal breadth first search. The complexity lies in constructing and updating the solution tables.

The purpose of the solution table in a node is to record all of the interesting intermediate proofs that terminate at that node. A given bitset is interesting if the intermediate proof for just that bitset would be different from that for some superset. When an attestation `edge` is traversed to reach a node `n`, the following procedure is followed to update `n`'s solution table

```
func (n *Node) UpdateSolutions(src *Node, edge *Edge) {
  askfor := make(map[uint64]bool)
  for _, sol := range src.Solutions {
    if sol.Bits&edge.Bits == 0 {
      //This solution doesn't go through the edge
      continue
    }
    askfor[sol.Bits&edge.Bits] = true
  }
  for _, sol := range n.Solutions {
    if sol.Bits&edge.Bits == 0 {
      //This solution doesn't go through the edge
      continue
    }
    askfor[sol.Bits&edge.Bits] = true
  }
  newsolutions := []*Solution{}
  for bits := range askfor {
    sols := src.BestSolutionsFor(bits)
    for _, sol := range sols {
      if sol.TTL == 0 {
        continue
      }
      nsol := sol.Extend(edge)
      newsolutions = append(newsolutions, nsol)
    }
  }
  allsolutions := append(n.Solutions, newsolutions...)
  pruned_solutions := reduceSolutionList(allsolutions)
  n.Solutions = pruned_solutions
}
```

In essence, for every bitset already in the destination node's solution table and for every bitset in the source nodes solution table, query the source node for the best solutions for that bitset (`BestSolutionsFor`), extend each one by the attestation being traversed, and add it to the destination node's solution table. Finally, prune all redundant solutions (`reduceSolutionList`)

from the destination nodes solution table by following the rules.

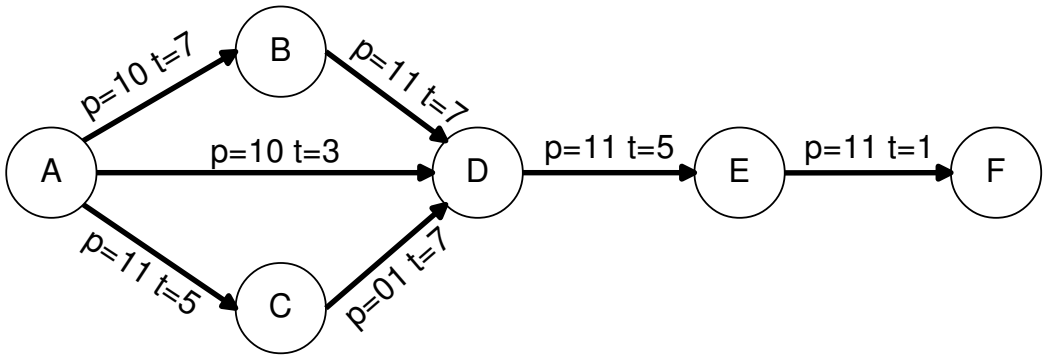


Figure 6.6: A scenario for solution tables

Consider Fig. 6.6. After completing depth 1, `B`'s solution table will have an entry for $10 \rightarrow \{([\overline{AB}], \mathsf{w} = 1, \mathsf{ttl} = 7)\}$. `D` will have an entry $10 \rightarrow \{([\overline{AD}], \mathsf{w} = 1, \mathsf{ttl} = 3)\}$. At the end of depth 2, `D`'s solution table will be $10 \rightarrow \{([\overline{AB}, \overline{BD}], \mathsf{w} = 2, \mathsf{ttl} = 6), ([\overline{AD}], \mathsf{w} = 1, \mathsf{ttl} = 3)\}, 01 \rightarrow ([\overline{AC}, \overline{CD}], \mathsf{w} = 2, \mathsf{ttl} = 4)$. Note that `D` does not have an entry for permission bitset `11` even though it can form that by combining other intermediate proofs.

At the end of processing a BFS depth, if any traversed attestations ended in the proving entity node, we perform a check for complete proofs. This calls `BestSolutionsFor` on the proving entity node with the full target bitset as the parameter.

The remaining function, `BestSolutionsFor` is responsible for deciding which entry or combination of entries in a solution table best realizes a given bitset. This function is not straightforward and we have multiple implementations that perform better under different conditions. All of our better performing implementations are based on solutions to the subset-sum problem. In the complexities, N is the number of entries in the solution table and k is the number of bits in the permission bit set (the number of statements in the policy).

Naive BestSolutionsFor - $O(2^N)$.. This algorithm forms a list of every subset of solutions in the solution table and returns all of the subsets that meet or exceed the required bitset. This runs in $O(2^N)$ but is guaranteed to find the optimal proof. The returned results will contain a lot of redundancy, which will be filtered out by `reduceSolutionList` which itself is $O(N^2)$. While the complexity on paper is abysmal, many real-world use cases might have small solution tables (<10 entries) so this algorithm might work fine.

Fast BestSolutionsFor - $O(Nk + Nlog(N))$.. This algorithm is a fall-back for when N or k is large enough that other algorithms are deemed too slow. This algorithm will find a solution if it exists, but will not return the lightest weight proof. It only has a simple, crude optimization for minimizing weight, and instead focuses on returning the highest TTL solution. The process is:

maxcount ← empty map from bitset to integer
for bitnum in 0 .. k **do**
    maxttl ← 0

```
        for sol in SolutionTable do
            if sol has bitnum set then
                if sol.TTL > maxttl then
                    maxttl ← sol.TTL
                end if
            end if
        end for
        for sol in SolutionTable do
            if sol.TTL = maxttl and sol has bitnum set then
                increment maxcount[sol.Bits]
            end if
        end for
    end for
    Sort SolutionTable by TTL descending and tie-break by maxcount
    result ← empty list
    resultBitSet ← 0
    for sol in SolutionTable do
        if sol has bits set that resultBitSet does not then
            add sol to result
            add sol.Bits to resultBitSet
        end if
        if resultBitSet = targetBitSet then
            return result
        end if
    end for
    return nil
```

Backtracking BestSolutionsFor.. This is a recursive algorithm for finding the best combinations of entries in a SolutionTable that provide the target bitset. The algorithm first pre-processes the solution table to form a set of distinct permission bitsets (so we ignore TTL and Weight at this stage). This is called `partialProofs`. We also then form a list where every element is the bitwise AND of all partialProofs with an index less than or equal to that element, called `sums`. We then recursively call:

```
    function FINDSUBSET(partialProofs, sums, target, curSol)
        if target == 0 then
            emit curSol as a solution
            return
        end if
        if partialProofs is empty then
            return
        end if
        if target > sums[-1] then
```

```
        return
    end if
    if partialProofs[-1] & target != 0 then
        FindSubset(partialProofs[:-1], sums[:-1], target & (∼ proofs[-1]), curSol + proofs[-1])
    end if
    FindSubset(partialProofs[:-1], sums[:-1], target, curSol)
end function
```

This function will emit every combination that achieves the desired target. We then re-introduce the TTL and Weight for each solution based on the TTL and Weight of its constituent partial proofs. If there were multiple TTL/Weight combinations for a given permission bitset, every solution returned by `FindSubet` using that bitset will be split into multiple solutions. We then call `reduceSolutionList` on the final list of solutions result to remove redundancy.

The final representation of a subgraph proof presents the set of attestations and a concise list of paths assembled from pointers to those attestations. This ensures the verifier of the proof evaluates all the permissions that the prover requires to be evaluated, without requiring an exhaustive evaluation of every path present in the subgraph.

6.10 Profiled Evaluation

We discuss the performance of the core cryptographic operations in Chapter 5, but these do not include the non-cryptographic overhead that stems from other aspects of the implementation. In this section we provide the end-to-end costs of the operations in WAVE, as well as providing a breakdown that shows what contributes to these costs.

These numbers are generated on Intel i7-8650U AMD64, the same hardware used in Chapter 5 but it is a different set of benchmarks so the numbers differ slightly.

Core Operations

This section instruments a WAVE daemon to determine which parts of each core operation are taking the most CPU time. We list the total wall time next to each heading, and then the percentage of CPU time for notable parts of the operation.

CreateAttestation - 78.3 ms. CreateAttestation is dominated by the generation of policy-specific RDE keys, which takes **89.3%** of the CPU time. Note that the key generation is spread over multiple cores, so this amounts to more than 560ms of CPU time on the 8-core CPU. After the RDE keys have been generated, **5.9%** of the CPU time is spent encrypting the attestation using the WKD-IBE key that permits decryption with discovered RDE keys. Additionally **1.7%** of the CPU time is spent performing IBE encryption, which is the outer compartment that ensures that the WKD-IBE ciphertext does not leak the partition. The time to perform CreateAttestation is heavily dependent on the number of cores available as the RDE key generation is highly parallelizable.

CreateEntity - 10.3 ms. CreateEntity is performed on a single core and is dominated by the key generation of the WKD-IBE key, which takes **76.0%** of the CPU time. Generation of the IBE

key takes **17.5%** of the total time. The generation of the Ed25519 and Curve25519 keys used for signing and public key encryption respectively take less than **1%** of the time. Serializing the public and private entity objects into their canonical form takes **3.5%** of the time. This is so high because the WKD-IBE and IBE keys undergo a somewhat complex marshalling procedure when converting from their in-memory representation to a compact serialized representation.

EncryptMessage - 6.1 ms. EncryptMessage will generate WKD-IBE and IBE encryption keys from a partition and the namespace public keys. This takes **10.0%** and **2.3%** of the time respectively. Once the keys are generated, the message is encrypted first with the WKD-IBE key, taking **67.2%** of the total time, and then encrypted by the IBE key, taking **18%** of the time. Usually EncryptMessage is used to encrypt a symmetric key to bootstrap a session and subsequent messages in the session use more efficient symmetric key cryptography.

DecryptMessage - 7.3 ms. Decrypt message is dominated by the cryptographic decryption operations. The IBE decryption takes **23.8%** of the time and the WKD-IBE decryption takes **49.3%** of the time. Loading the keys from the database to support decrypting these ciphertexts takes **14%** of the total time, which is largely the deserialization of the marshalled keys.

UpdatePerspectiveAttestionDirect - 66.2 ms. This time applies when an attestation is granted directly to the proving entity, and they perform an update of their perspective graph. Loading the entity objects while updating the perspective graph takes **29%** of the time. Only **15%** of the time is spent parsing and decrypting the new attestation. **23%** of the time is spent unmarshalling the RDE keys out of the attestation and an additional **25%** is spent indexing those keys and inserting them into the database.

UpdatePerspectiveAttestationDelegated - 80.7 ms. The same process as in the directly-granted attestation is followed, and all of the tasks accounted for there occur in the delegated scenario as well. The additional 18ms are split evenly between loading the RDE keys from the database to support decrypting the attestation, and actually performing the IBE and WKD-IBE decryption operations. The time to update a perspective graph with an arbitrary number of additions can be estimated well by multiplying the number of new attestations by 80ms.

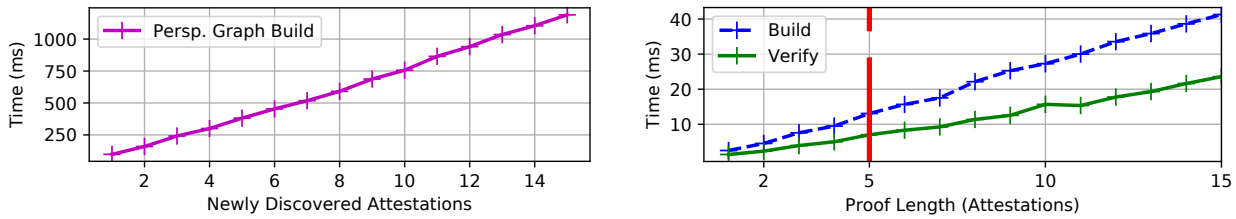
These end-to-end costs, especially for decrypting new attestations in the perspective graph, are quite a bit higher than the cost of the core cryptographic operations. Recall from §5.8 that the WKD-IBE and IBE decryption operation takes just 6.2 ms, under a tenth of the total end-to-end cost of updating the perspective graph with a new delegated attestation. We do not believe these overheads are intrinsic, and could potentially be reduced with changes to the implementation. One candidate is the moving away from the use of Gob[60] encoding when placing objects into the perspective database. This is a convenient encoding format as it can represent complex Go types, including interfaces and substructures, without any additional code. It is not the most efficient, however. If a special-purpose representation of entities, attestations and keys were made just for their storage in the local database (distinct from their canonical DER representation), we could likely reduce the time spent performing serialization and deserialization. One option might be using a higher performance less-feature-rich encoding, such as MsgPack [59], at the cost of increased code complexity.

In practice, these operations are already fast enough that normal use is not encumbered by them. Changes to the permission graph are normally triggered by human actions, and the 80ms it

takes to process a new attestation is far less than the tens of seconds it takes a human to grant new permissions, meaning these operations are not the bottleneck.

Proof Building and Verification

As discussed in §3.8, discovering new permissions is the slowest operation in Graph Based Authorization. The time taken to update the perspective graph is linear in the number of new attestations, and is generally a multiple of UpdatePerspectiveAttestationDirect and UpdatePerspectiveAttestationDelegated, depending on if the attestation is one hop (direct) or more (delegated), with a network round trip per attestation to retrieve the ciphertext from ULDM storage. This is shown in Fig. 6.7a. It can be seen that it takes about 750 milliseconds to add ten new attestations to the perspective graph, and this includes all cryptographic operations and indexing of the new keys. We believe that this is fast enough to not pose any problems, considering that attestations are typically created as a result of human-triggered events that take place at much lower frequencies.



(a) Perspective graph build

(b) Proof build/verify

Figure 6.7: Vertical line in Fig. 6.7b is the expected maximum proof length for common applications.

Proof building is also linear in the size of the proof, assuming the topology of the graph allows for pruning of the paths that are traversed. It is possible for proof building to have very different complexities depending on the topology of the graph, as one can see from the discussion in §6.9. Proof building times for different sizes of linear topology are shown in dashed blue in Fig. 6.7b. In our deployments we have found that most proofs are less than five delegations long, and this is shown by the vertical red line. It can be seen that even proofs with five delegations complete in under 15ms which is comparable to the latency of the sequence of network round trips that would occur in traditional authorization systems.

Proof verification is always linear in the number of constituent attestations. The time taken to verify a proof with varying numbers of delegations is shown in solid green in Fig. 6.7b. Proof verification is very fast, completing in under 10ms for proofs with five delegations.

Traditional Authorization Flow

To compare WAVE against a traditional authorization system, we benchmark the time taken by a representative Discretionary Access Control backend to turn a username and password into an au-

thorization policy using an OpenLDAP server (which authenticates the user and yields the groups they are part of) and a MySQL database (which turns the groups into policy). Both the LDAP and SQL servers are available to the backend with negligible network latency (less than 0.1ms).

The results are shown in Table 6.1. For a WAVE proof mirroring the single-delegation structure present in the LDAP/OAuth2 case, the proof verifies in a sixth of the time taken by the traditional LDAP flow. For a case where transitive delegation has been used three times and the proof consists of three attestations, the WAVE verification is about half the time of the single-delegation LDAP flow.

System	Authentication	Authorization
LDAP+MySQL	6.3ms	0.8ms
OAuth2 JWT	0.3ms	
WAVE 1 attest.	1.2ms	
WAVE 3 attest.	3.6ms	

Table 6.1: Latency of LDAP+MySQL, OAuth2 vs. WAVE.

We also add the time taken to verify an OAuth2 JWT token containing the authorization policy in the form of scopes. As in WAVE, OAuth2 offers a bearer token that can be validated without communicating with the server. In this case, validating a JSON Web Token with a 2048-bit RSA signature takes 0.3ms. WAVE is roughly 4x slower, but completely removes the centralized token-issuing server, leaving the user as the only authority in the system. In OAuth a compromised token issuing server can generate valid tokens without the user's knowledge.

Note that although OAuth2 has added a form of delegation [71], it requires the OAuth2 server to issue a new token, so is identical to the single-delegation scenario tested here.

This example shows that using WAVE as a replacement for common authorization flows will likely not reduce performance, despite providing transitive delegation and removing all central authorities.

6.11 Summary

We have constructed a fully working prototype of WAVE, with an implementation of RDE (Chapter 5) and ULDM storage (Chapter 4). Although there is definitely room for improvement in the implementation, even this prototype offers performance numbers that are comparable to leading commercial authorization systems. This demonstrates that graph based authorization is a feasible alternative to centralized systems.

Future work on the implementation may focus on caching at different layers of the system, and on improving the efficiency of serialization and deserialization when writing to and reading from the perspective graph in local storage.

As discussed in §5.7, there are advantages to introducing KP-ABE as an alternative to WKD-IBE for RDE. This should be straightforward given the use of ASN.1 EXTERNAL types identified

by OID within WAVE objects and should be able to be done in a way that allows existing attestations to remain valid.

Chapter 7

Microservices in the Built Environment

Although presented in isolation in the previous chapters, the initial use case that led to the design of WAVE was authorization in the built environment. The built environment is an excellent example of a distributed system spanning multiple authorization domains with multiple tiers of administration and a variety of devices and services that require fine-grained access control mechanisms. It is also an area where security has not been well addressed, with several high profile breaches of building control systems occurring over the past few years [77]. This chapter presents a microservice-oriented extensible building operating system based on WAVE that helps to unify authorization and management. This chapter appears in more detail in [6].

7.1 Building Operating Systems Background

Where traditional Building Management Systems (BMS) in commercial buildings provide a central point of supervisory control that is connected to and providing set-points for many dedicated direct controllers and provide a limited set of services (status screens, schedules, logging, trending, alarms), Building Operating Systems (BOS) seek to provide richer functionality, flexibility, extensibility, and federation. Rather than a separate system for HVAC, lighting, etc., these are integrated in a common BOS. New systems, such as electrical usage monitoring [39], environmental or CO2 sensing [106], or appliance control [30], are often further integrated with these conventional building subsystems to support new operating modes, such as demand response or demand controlled ventilation.

Over the past few years, building operating systems have converged on an architecture similar to that shown in Figure 7.1. This has a set of services acting as a hardware presentation layer for a heterogeneous set of hardware accessed over a variety of interfaces. The HPL serves to promote different devices to a common communication format on a bus. Higher level services, including archivers for time-series data, metadata stores, query processors, controllers, arbiters, and schedulers attach to the bus (as additional persistent processes) and support portable applications, including occupant-centered conditioning, energy analytics, diagnostics, prognostics, model predictive control, and so on. These applications are also essentially persistent processes dropped

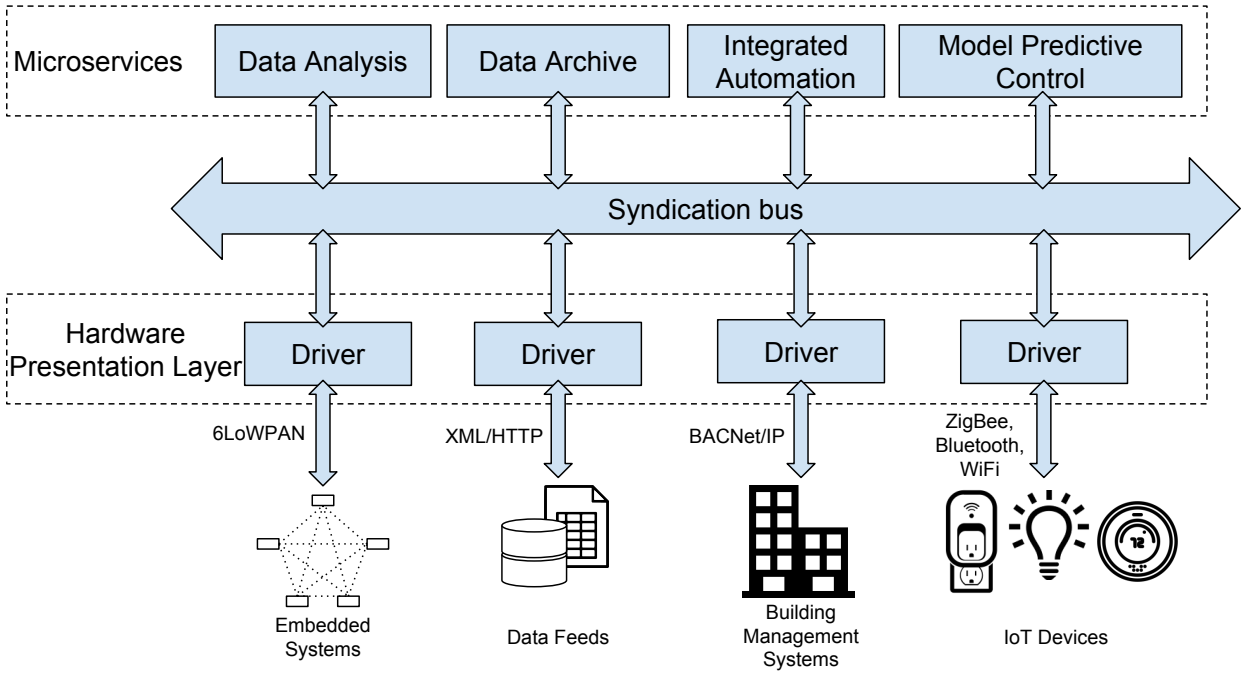


Figure 7.1: Common Building Operating System Architecture

into the cyber-physical building distributed system, where they can be accessed in turn by services performing data analysis, automation and storage.

Despite the convergence at other layers, there has not been an agreement on how security and authorization are implemented. Some systems implement security in the broker/archiver (e.g., Mortar.io [93], Sensor Andrew [96], HomeOS [42]), some have distinct security models for sensing vs actuation (e.g BOSS [38]), some assume applications are fully trusted (e.g VOLTTRON 2.x [4]) and many simply rely on the applications to implement security.

A decentralized authorization, system such as WAVE, is a good match for building operating systems as it allows the intrinsically distributed system to avoid the use of a centralized authority. Such a system preserves the converged architecture of Fig. 7.1 while improving security and manageability.

Hardware Presentation

The many distributed sense and control points in the built environment are on a variety of different interconnects (RS485, BACnet, Ethernet, WiFi, LoWPAN, etc.) with a variety of different access methods and protocols. To unify these, BOS wrap these devices with persistent processes acting as a hardware presentation layer. This layer of abstraction, which centers on the common notion of a persistent process acting as a device driver, has taken a number of similar forms in prior work. For example, HomeOS [42] features service interfaces, Beam [101] has adapter modules,

and BOSS [38] has sMAP drivers [39]. In all these cases, the drivers communicate over a common syndication layer for device and service discovery, data reporting, and actuation.

Unlike application services, these drivers usually have restrictions on where they can run. For example they may need to be in a particular location that is either physically connected to existing building communication media (e.g., BACNet), or inside a local network (e.g., to connect to a local smart thermostat that exposes an insecure HTTP interface).

Syndication

The syndication tier is the backbone in Figure 7.1 that connects all services, drivers and applications. The majority of BOS have converged on the publish/subscribe resource-oriented communication paradigm within this syndication tier as it offers advantages such as location transparency, easy multicasting of messages between arbitrary and dynamic collections of producers and consumers, and a clean abstraction for event-based programming. In BOSS [38] and sMAP [39] the pub/sub broker is strongly coupled with the data archiver, and is used for sensing but not all forms of actuation. Similarly HomeOS [42] and Beam [101] both have a single server performing message dispatch and archiving, although the semantics of the communication channels differ. Sensor Andrew and Mortar.io use XMPP [115] which is closest to Figure 7.1 as the broker performs purely syndication and authorization, leaving data archival as a distinct service.

Note that in all of the above systems, the syndication mechanism is also a centralized authorization authority, which we can use WAVE to avoid.

Data Archive

Most building operating systems gather and store sensor data for applications, such as energy data analytics and occupancy-based environmental control. Storage of and access to this data is typically managed by a central authority, normally the building administrators. To deploy a sensor that reports building data or an application that consumes sensor data, one has to obtain permissions from the central authority. This pattern is present in BuildingDepot [2] and in BOSS [38] for example. Some approaches distribute the data (e.g., Mortar.io [93, 27]) but employ a centralized authorization system.

It is necessary, as we expand from the building to the built environment, to have an architecture supporting multiple archivers and storage systems, owned and managed by different people.

Applications

Much of the functionality of a building operating system is derived from the set of persistent services and applications it hosts. Some of these perform analytics on data to create new streams of information (e.g., anomaly detection [56] and occupancy sensing [9]), some of these perform control of building processes (e.g., demand response [9] and model predictive control [72]) with the aim of reducing power consumption or improving occupant comfort, and some provide miscellaneous services (e.g., visualization services).

7.2 An eXtensible Building Operating System

WAVE provides security primitives, but by itself does not act as an operating system. XBOS, the eXtensible Building Operating System, is built on top of WAVE and fulfills the role of a building operating system by using *microservices*: small single-purpose persistent processes. The notion of a microservice architecture for large scale distributed system design is already well known and well proven. The contribution here is the synergy between microservices and the authorization model of WAVE. Core to this system is the development of a new syndication tier that uses WAVE to provide decentralized authorization. This is in contrast to the centralized ACLs used in most existing publish/subscribe systems, such as XMPP [115] or MQTT [13].

Secure Syndication

Security in XBOS is predominantly enforced at the syndication layer. We discuss the details and advantages of this particular architecture in Chapter 8, but the synopsis is that is allows heterogeneous services and devices to be managed using fairly uniform security policies. To enable this, we construct a publish/subscribe system, WAVE Messaging Queues (WAVEMQ) that integrates WAVE for security. Specifically, what are typically referred to as topics in publish/subscribe systems, become WAVE resource URIs. This means that the RTree policy type can express exactly to which streams of messages a user can publish or subscribe. WAVEMQ will also act as a WAVE daemon, building proofs for users, attaching proofs to every message, and validating the proofs for received messages. We discuss WAVEMQ in detail in Chapter 8.

An additional aspect of WAVEMQ that is relevant here, is that by enforcing WAVE policies in the broker (or *router* as it is called in WAVEMQ), we ensure that unauthorized messages do not get transmitted to clients. This makes it harder for an attacker to mount a Denial of Service (DOS) attack against services communicating using WAVEMQ. This property is explored more fully in the next chapter.

Containerized, Persistent Services

Underpinning XBOS is a solution to an oft overlooked issue in microservice architectures and in building operating systems in general: where do these services run and how are they managed? Concretely, this equates to six questions, driven by service lifecycle:

1. How does a service obtain permissions to consume its input resources and produce its output resources?
2. How does a service obtain its initial configuration?
3. How does a service get scheduled to run?
4. How is the service isolated such that failure does not couple to colocated but unrelated services?

5. How is a service monitored and how is authority to monitor it obtained?

6. How is a service retired?

Resolving these questions at a low level can drastically reduce the complexity at higher layers. The status quo, even on embedded devices, is to have a Linux device on a trusted network and create accounts for each administrator allowing them to run processes that are then managed over SSH.

This works well at a small scale, but complexity quickly gets out of hand. If a single admin is compromised, that account has full access to all resources on the local network. Services can use too much memory or CPU time, affecting other services. Remnants of old services accumulate and dependency conflicts (e.g., versions of python libraries) complicate new service deployment. Integration is done by sharing SSH credentials obscuring who and what has access at any given time. These problems, while individually insignificant, as a whole contribute to a high management overhead and limit the scalability and usefulness of the overall system.

Spawnpoint is a ground-up solution to these problems. Its purpose is to deploy, run, and monitor the microservices, which together form an instance of XBOS, within managed execution containers. Thus, Spawnpoint can be thought of as the "proto-service" combining Docker and WAVE upon which all other microservices are built. It begins with the realization that the resources required for persistent processes (e.g., CPU time) are the same as any other resources (e.g., sensor data) and can be managed using the same WAVE and WAVEMQ primitives. The ability to deploy and manipulate microservices is tied to the ability to publish commands on certain WAVEMQ URIs, while the visibility of execution containers, as well as the computational resources that back them, is contingent on the ability to subscribe to messages on certain WAVEMQ URIs. Spawnpoint enforces isolation between running services and also performs admission control to ensure that resources do not become oversubscribed.

A microservice to be deployed into a Spawnpoint is described declaratively in a manifest that encompasses:

1. What code to run

2. What environment it needs (libraries and version etc.)

3. The configuration parameters

4. The isolation parameters (CPU time, memory, network access, etc.)

5. Placement restrictions (requiring a particular architecture, or particular network)

The microservice is then instantiated simply by using WAVEMQ to publish this manifest to a URI representing a Spawnpoint instance that satisfies the necessary placement constraints. This instance is responsible for managing a specific collection of computational resources (e.g., an on-premises server or a cloud-based virtual machine). Note that the publication of this manifest is no different from any other WAVEMQ operation, so a WAVE proof is generated by the sending WAVEMQ

daemon and validated by the receiving WAVEMQ daemon in spawnpoint. Upon receipt of a new service manifest, the Spawnpoint daemon builds a docker container with the correct environment and code. It injects the configuration parameters (including the WAVE Entity that represents the microservice) and launches the container, using cgroups to enforce the isolation parameters. Various monitoring metrics, such as CPU usage, memory usage and program output, are streamed to WAVEMQ URIs representing that microservice. Monitoring and logging of microservices is thus accomplished through WAVEMQ subscriptions.

This approach borrows heavily from microservice systems, such as Kubernetes [57], but is distinguished by the use of WAVE and WAVEMQ for the protected control interface (rather than a REST API as in Kubernetes). Spawnpoint makes creating, manipulating, and monitoring a microservice no different from any other operation on a WAVEMQ URI. Therefore, Spawnpoint inherits all of the permission verification and delegation benefits of WAVE without any complexity added to its own implementation. In particular, this gives a user the ability to independently and locally delegate partial access to the services they control, and manage that delegation. In addition, it adds transparent and effective DDOS protection to services, a problem usually poorly mitigated by costly over-provisioning.

This ability to partially trust services (with fine grained control over what they have access to and what resources they use when running), allows us to run third party applications that we do not fully trust. This is a critical feature provided by computer operating systems for decades, but is often ignored in the built environment or IoT context. One could imagine that building monitoring and alerting would be an off-the-shelf third party application that you install on your building. You vaguely trust the application to do its job, but that does not mean you should be forced to trust that it will not attempt unauthorized actions or that you must audit the application yourself (e.g., in VOLTTRON [4]).

7.3 A Secure Building Operating System

Using WAVE for authorization, WAVEMQ for communication, and Spawnpoint for service deployment and management, it becomes possible to efficiently provision and maintain applications for the built environment as collections of microservices running on distributed infrastructure. A traditional approach to authority in this setting would require administrators to create SSH credentials for each service developer on each machine they wish to use. This cannot scale to a large number of machines or users and is cumbersome to maintain over time. Instead, WAVE's permissions graph can dictate who is allowed to deploy code to various Spawnpoint-managed machines, abstracted as WAVEMQ URIs. This gives rise to a more flexible and dynamic model where microservices are rapidly deployed and can even be provisioned on-demand, such as when a new occupant enters a building.

XBOS, the eXtensible Building Operating System, fulfills the role of a building operating system by using microservices deployed with Spawnpoint. XBOS's microservices include device drivers, services for data cleaning, and adapters to either transform data or wrap software interfaces to ensure compatibility. These services are each pushed to a Spawnpoint instance, poten-

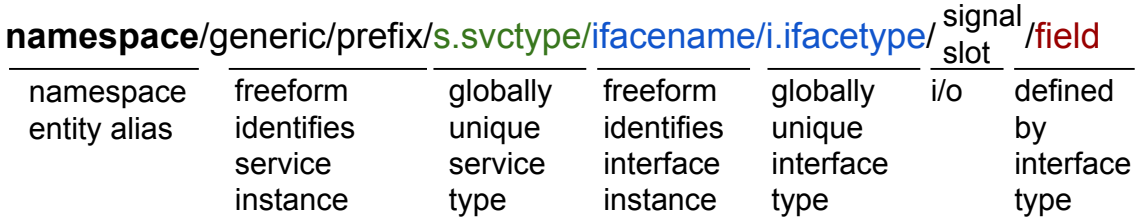


Figure 7.2: The XBOS URI structure capturing services and interfaces to enable autodiscovery

tially running in the cloud or on premises, where they interact and form compositions with other microservices via communication over WAVEMQ.

XBOS services and interfaces

While WAVE (and consequently WAVEMQ) permits resource URIs to have any form, we have found it convenient to create a set of conventions on URI format as shown in Figure 7.2. These conventions then permit service and interface discovery, as well as metadata propagation.

A *service*, in the abstract overlay sense, is a logical grouping of interfaces. A single physical device, such as a thermostat, would be a service. A controller or scheduler would also be a service.

Within a service, there are multiple *interfaces*. A service may implement multiple interfaces and these may offer overlapping functionality. For example a thermostat may offer a temperature sensor interface alongside a generic thermostat interface and a vendor specific thermostat interface. This allows applications designed to consume temperature sensor data to interact with that part of the thermostat, without needing to know anything about thermostats.

The interfaces composing a service do not need to all be implemented by the same running process. If an application requires an interface that a service does not expose, it is common to spawn an adapter process that consumes existing interfaces and refactors them into a new interface for purposes of interoperability.

Interfaces are broken into *signals* which are resources emitted by the interface, and *slots* which are resources consumed by the interface. A thermostat would have a slot for changing the setpoint and a signal for the current temperature, for example.

Metadata can be attached to services and interfaces by using persisted messages in WAVEMQ. This metadata provides additional context for the service or interface. For example it may convey that the thermostat service represents a device in a particular room in a particular building.

The format of this metadata is not constrained by WAVEMQ, but again some conventions help with interoperability. Within XBOS we are using the Brick schema [11].

Types of microservices

Following the converged architecture, XBOS is composed of three categories of microservices, which could potentially be drawn from existing efforts.

Driver

A driver serves to elevate the existing interface of a device to a WAVEMQ interface within a WAVEMQ service. This interface is often an insecure local area network connection which restricts the placement of the driver to the same local network as the device. The device is then firewalled to only allow communication with the Spawnpoint on that network. This ensures that the WAVE security policies cannot be bypassed by going directly to the device.

The notion of a driver performing hardware abstraction has existed in almost all prior work in operating systems for the built environment. The improvements here in security and ease of management are inherited from WAVE and Spawnpoint.

Analysis and adaptation

Many applications in the built environment act to take some input data, transform it, and produce some output information. We can assume that the the inputs and outputs are WAVEMQ resources as the driver infrastructure takes care of the protocol adaptation required. Therefore there are no placement restrictions on these services. It may be beneficial to run these on a platform where computation is cheap (for example the cloud), especially for heavyweight analytics like computer vision and machine learning.

In the majority of cases, the security policy of the application can be completely expressed as a set of permitted input resources and permitted output resources, e.g., what the application can see and who can see what it produces. In some cases (e.g., in [10]) more complex policy is required, such as ”X can see office occupancy only during work hours, or X can see aggregate information about a floor, but not individual offices”. This can be implemented by spawning a policy adapter microservice that consumes the raw information, validates it against the policy, and publishes information in accordance with the policy. The end user is then granted permission to consume only the output of the policy adapter, not the raw data. This allows arbitrarily complex logic to be enforced.

Controllers

A controller consumes a set of sensing and parameter resources and writes to a set of actuation resources. Examples include things like setpoint schedules, setbacks, etc. As above, there are no placement restrictions, which allows complex controllers such as model predictive control, to be executed where computational resources are cheap.

Heavyweight services

Some services are more heavyweight and do not benefit much from the platform abstraction that Spawnpoint provides. A good example is an archiver. For a large deployment, the archiver will likely have a dedicated server with several RAID arrays. In addition, there is typically a static globally-routed IP address associated with the service. At this time Spawnpoint is not designed to offer this, so these services can be deployed outside Spawnpoint.

This does not affect the interfaces exposed over WAVEMQ - consumers are always indifferent to the location and platform a service is running on - but it does mean that health monitoring and administration tasks such as upgrades must be done using conventional tools and methods.

7.4 Summary

This chapter presents a brief overview of a secure building operating system that leverages WAVE for authorization policies and WAVEMQ for syndication. There are many benefits to such a building operating system that are out of scope for this dissertation, but a key contribution that is relevant to our goal of decentralized authorization is the hardware adaptation layer. The architecture of XBOS allows us to adapt the heterogeneous interfaces for devices within the built environment, presenting a uniform set of interfaces upon which a homogeneous security policy can be defined. We discuss this further in Chapter 8.

Chapter 8

Secure Syndication for IoT

Throughout the development of XBOS, we discovered first-hand that managing distributed IoT systems is a complex task. One of the major sources of complexity is ensuring that all the devices and services are appropriately secured, especially as the set of authorized users evolves over time. This management can be simplified by having a unified security model so that administrators can reason about a homogeneous set of security policies where similar actions across similar devices or services are granted in the same way. In addition, we can reduce the room for error by having a single record of truth for authorization, as opposed to a fragmented set of device-specific security policies where some devices may not support a policy mechanism expressive enough to reach the desired security goals. For example, many IoT devices have an authorization interface consisting of only a single username and password. As a result, the only policy they can affect is a boolean one of no-control vs absolute control. This is also further complicated by the fact that such an authorization mechanism does not map well onto patterns of authorization that include delegation.

One method for increasing the homogeneity of security policies is to apply security at a higher layer of abstraction than the raw device functionality, thereby sidestepping the limitations of device-specific authorization functionality. As shown in Fig. 8.1, as abstraction increases, so does the uniformity of security policies. At the extreme level, every device is abstracted to a single boolean specifying if a user can use the device or service (all access to all functionality) or if they cannot. The goal is to identify the right amount of abstraction so that similar devices or services can have similar policies applied to them but security policies retain the granularity that matches the user's requirements.

This pattern applies to services, too. A low level of abstraction for a service would be an application-specific policy that reasons about constraints and requirements that are not applicable to other applications. This makes it hard to write cross-service authorization policy. A syndication-level policy would specify at a higher level, deciding which RPC functions can be invoked, for example, assuming that the mapping from service to resource is done at the function level. If the service adheres to common interfaces (as is the case in XBOS), then services of a similar type can have identical authorization policy applied to them, even though they may be implemented in different ways.

The syndication layer appears to be this ideal location. In the XBOS architecture, all interac-

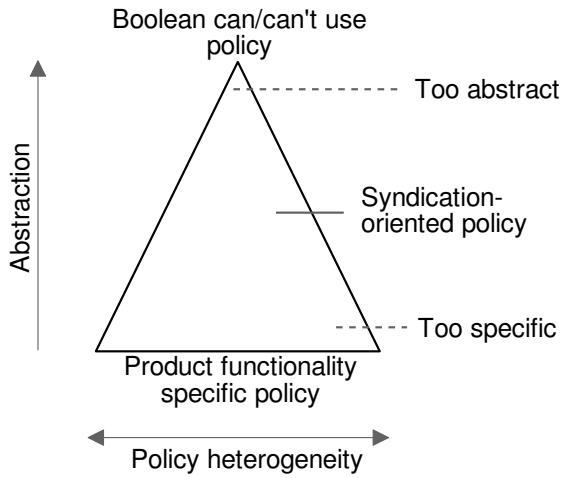


Figure 8.1: The design space for authorization policy abstraction

tions with devices and services occur through this syndication layer, so if security is applied here, it carries through into the security of the device as a whole. This assumes that other methods of access to the device (e.g. physical or other digital connections) are made unavailable or are secured through other means.

As we discuss in §7.3, devices in XBOS have a microservice adaptation layer in front of them that can present the control and sensing interfaces of the device as resources compatible with the syndication layer. In our case, these microservices present control and sense points as publish/subscribe resources.

This chapter presents the design of the WAVE Messaging Queues secure publish/subscribe syndication layer for IoT.

8.1 Resource Design For Syndication Security

The microservice representing a device presents resources that can be published to for actuating the device and resources that can be subscribed to for consuming the sensor data the device generates. If we apply a security policy at the syndication layer that filters out unauthorized publish or subscribe commands, then services and the devices they represent can operate with a simple policy of "if I receive a command, it is authorized". This simplifies the programming of devices and services. This also, for better or for worse, matches the security policy of many IoT devices on the market that rely on a secured network and do not implement any of their own access control.

The granularity of the security policies in this approach is then determined by the granularity of the communication interfaces. Consider a device service that exposes just two interfaces: `/sensors` and `/control`. Such a service allows permissions to be granted independently for access to the sensor data and actuation of the device, but that is the extent of the granularity. The policy can not differentiate between the different types of commands that are sent to `/control`

or the different sensors that are published on `/sensors`, at least not without the service observing the proof and performing its own additional authorization, which we are attempting to avoid.

Instead, a properly segregated service might publish sensor data on distinct resources such as `/sensors/temperature` and `/sensors/occupancy` and have unique control points for each type of command or piece of adjustable state such as `/control/setpoint` and `/control/schedule`. This allows syndication-level policy to express much finer granularity, where one might, for example, grant access to see environmental conditions but not personal activity. This segregation offers other non-security benefits as well, such as the ability to subscribe to just the data you are interested in, reducing network traffic, so the designer of a service is motivated to use distinct resources even if not considering the security benefits.

Therefore, to allow for expressive security policies, we construct our microservices to have distinct resources for each set of sensing and actuation points that we would grant access to. It is acceptable to bundle sensing points into a single resource if permissions on those sensing points will always be granted together. Fig. 7.2 in the previous chapter shows the structure of resources in XBOS that adheres to this design pattern.

8.2 Denial Of Service Resistance

Publish/Subscribe systems can be implemented in two major ways: with a broker, such as MQTT/AMQP, and without, such as ZeroMQ. When considering the effects on security, there is an advantage to the broker-based design: the devices in the system are only communicating with one host at Layer 3 – the broker. This allows very simple firewall rules to prevent any other network traffic from reaching the device being secured. With a brokerless design, such as ZeroMQ, traffic flows directly between participants so the the firewall rules need to be adjusted every time a new participant needs to communicate with the device.

The importance of static network policies as allowed by the broker design cannot be overstated. In this design all communication occurs with the broker and all other traffic is rejected. The more dynamic evolution of authorization policies can then take place using WAVE, without requiring additional interactions with the network firewall appliances. For example, if a new controller is installed that requires access to several sensors, no changes need to be made to firewall policies. In contrast, good security using the brokerless design or standard peer-to-peer model would require constant updates to firewall rules for each pair of communicating services and devices. As a result, administrators tend to favor the less secure but easier to maintain "secure network" model which allows all devices and services within the "secure network" to communicate with each other (e.g. [4]). As discussed in §3.8, this approach is at odds with new trends in network security such as BeyondCorp [111] which argues that a "secure network" is not achievable. Enforcing network security only at the perimeter was the reason that Target was hacked through its HVAC system [77]. WAVE offers an alternative that we believe is equally easy to administer with significantly better security characteristics.

This is one of the ways that careful engineering of the syndication system can reduce the management overhead of a distributed IoT system: reducing the amount of configuration that

must be manually kept in-sync by system administrators. For this reason we exclusively consider the design of a broker-based syndication system, although WAVE can be applied to all forms of syndication.

In a broker-based publish/subscribe system, we are presented with another opportunity to increase system security. In addition to verifying the WAVE proofs at the endpoints (the recipient of messages), we can additionally enforce the security policy in the broker itself. We do not wish to trust the broker (which would constitute a central point of weakness) so we will need to continue verifying the proof attached to each message at the endpoints, but additionally enforcing the policy at the broker has an advantage: denial of service attacks are much harder to mount. Unauthorized messages sent by an attacker to an IoT device will be filtered out at the broker level, rather than making their way to the device and being dropped there. IoT devices are often resource constrained in multiple ways, having limited CPU power, operating over low-bandwidth wireless connections and in some cases even running off battery power. This means that even a modest amount of traffic could easily overwhelm it. In contrast, the broker is running in the cloud where it is easy to dynamically increase capacity in response to increased load. By ensuring that no unauthorized traffic reaches the end devices we limit the attack surface considerably.

Note that in order for policy enforcement at the broker to be effective, all other channels to the resource constrained device (such as direct IP connections) must be secured by other means, such as a firewall placed before all bandwidth-constrained links.

Syndication-level security policies lend themselves well to enforcement at the broker, as the information the broker requires to place messages in the correct queues (the resources) is the same as the information required to enforce the security policy.

8.3 Intermittent Connectivity

A property of legacy building control systems that remains valuable as we move forward towards IoT replacements is that of local control: if a system loses internet connectivity, it should continue to operate with as much functionality as possible. As a simple example, if a residential building loses its internet connection, that should not affect the operation of the lights or thermostat within the building. Yet, when these devices *are* connected, they should be able to make use of external resources.

The typical publish/subscribe architecture has a single broker that relays all messages in the system. In the multi-building case, it is likely this broker is located somewhere central, such as a cloud datacenter. Consequently, a loss of internet connectivity at one building would prevent all traffic from flowing between devices and services at that building, even though they still have a network connection between them. Even when not considering failure, it is desirable to limit the traffic that must flow out of the building just to be redirected back into the building to a service on the same network.

The solution is a tiered publish-subscribe system where there are site-local brokers that can relay local traffic between devices and services within the site, and a higher-tier central broker that can relay traffic between sites.

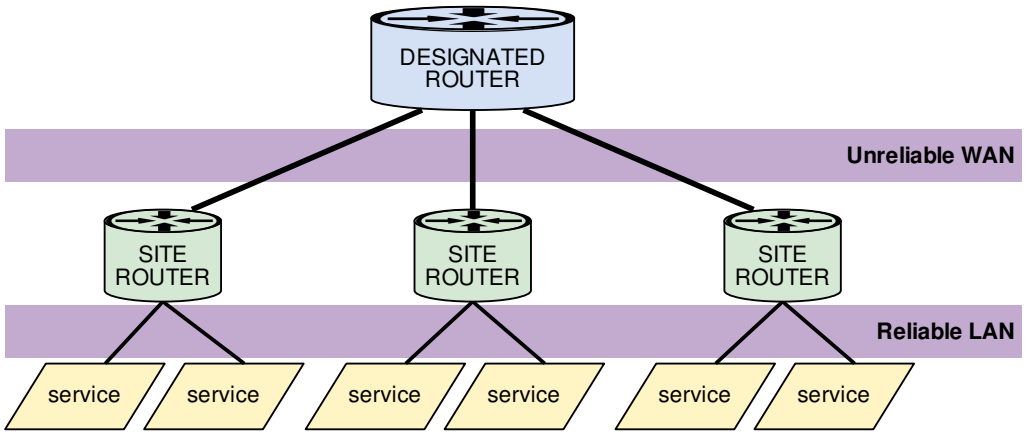


Figure 8.2: The topology of a WAVEMQ deployment.

Additionally, these brokers should perform queuing so that, where possible, a temporary loss of internet connectivity at a site does not result in a loss of data being streamed to off-site monitoring systems but rather just a delay in the data.

When searching for an authorization system that matches this model of potentially-disconnected site brokers and non-interactive message delivery, one quickly realizes that the authorization systems that are in widespread use do not cleanly apply in this case. Centralized ACL based systems would not permit site-local traffic to be authorized during internet outages and bearer-token based authorization systems would not work well with delayed message delivery as the tokens may expire while the message is in the queue.

In contrast, WAVE works well in this setting. WAVE proofs remain valid until a constituent attestation is revoked or expired, unlike short-lived bearer tokens. Furthermore, WAVE proofs are verifiable without any communication with an external service (with the exception of the optional revocation checks). We develop a WAVE-based tiered publish/subscribe system called WAVEMQ.

8.4 WAVEMQ System Overview

WAVEMQ combines WAVE's proofs and Resource Tree authorization policies with tiered publish/subscribe message routing. As shown in Fig. 8.2, local message delivery is performed by a *site router* and inter-site message delivery is performed by a *designated router*. We name this a designated router as it is the router that has been designated as canonical for a specific WAVE namespace.

The architecture of a WAVEMQ router is shown in Fig. 8.3. At the core lies message routing. This component receives authorized messages and distributes them to peer connections, and local connections, through queues. Local connections exist only on site routers and are established with devices or services. The peer server component exists only on designated routers and receives uplink and downlink peering requests from site routers.

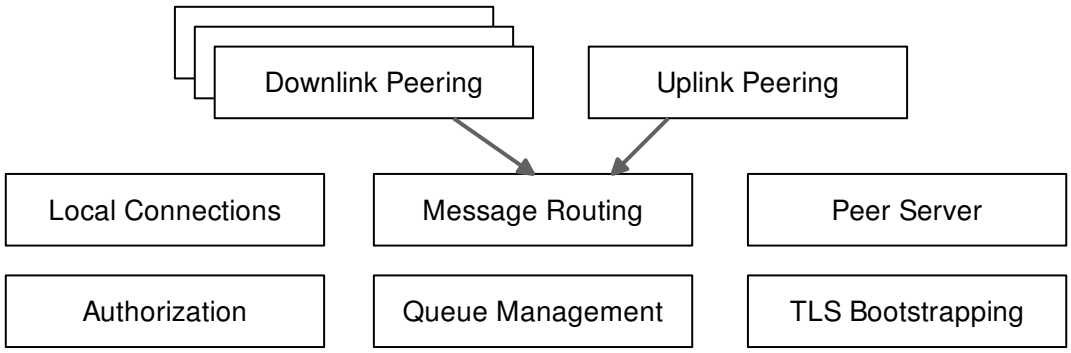


Figure 8.3: The components of a WAVEMQ router.

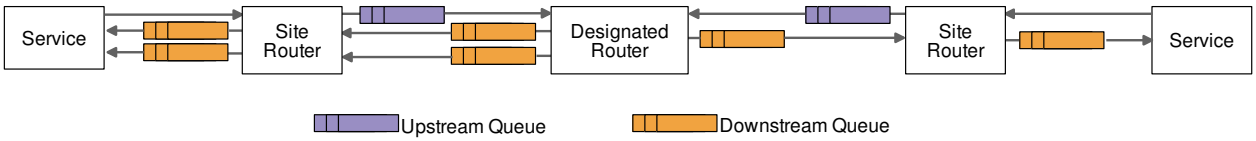


Figure 8.4: The queues in WAVEMQ

All peering connections use a unique TLS bootstrapping mechanism that uses ephemeral self-signed certificates for TLS, but then includes a WAVE signature of the self-signed certificate, along side a proof that the entity that made the signature is authorized for message routing. For example when a site router connects to the designated router, the designated router will return a signature over its ephemeral certificate made with the router's entity, and also return a proof that the namespace has granted permission to the router's entity to route traffic for the namespace.

This mechanism allows us to use TLS for transport security without relying on the Certificate Authority infrastructure which is a central authority. The compromise of a CA allows for the generation of seemingly valid certificates for arbitrary domains, which has happened several times [63, 62, 61]. Our bootstrapped TLS certificates do not suffer from this vulnerability.

As shown in Fig. 8.4, WAVEMQ uses unified upstream queues (shared among all resources within a namespace) to buffer messages from the site router to the designated router. In contrast, every subscription has a unique downstream queue so that different subscriptions do not interfere with each other. The service on the left in Fig. 8.4 has two subscriptions, so the site router has two queues for it. There are two matching queues in the designated router for these subscriptions as well. Services do not have upstream queues: it is assumed that the network connection between services and site routers is reliable. This is also motivated by the desire to keep the implementation of device services simple and place all complexity within the syndication service. Downstream queues exist at the site routers because we assume that services themselves are not reliable and may restart or temporarily go offline.

8.5 WAVE in WAVEMQ

When a service connects to a WAVEMQ site router, that site router also acts as the WAVE agent for the service. The service does not interact with WAVE itself, instead proof generation and verification is done by WAVEMQ on behalf of the service. The service is typically run on the same machine as the site router so that secure IPC can be used for communication.

For example, when a service abstracting a sensor publishes data, it will send its entity private key to the WAVEMQ site router along with it's request to publish the message. Internally, WAVEMQ will determine the proof that is necessary to show the message is authorized, and it will use WAVE to generate the proof on behalf of the service. This proof is then attached to the message, and the pair is sent to the designated router. The designated router will verify the proof and drop the message if the proof is invalid. If the proof is valid, the message and proof pair will be enqueued for delivery to various site routers. When the site routers receive the message/proof pair they will re-verify the proof. This re-verification ensures that a compromised designated router is unable to generate or forward invalid traffic that will affect clients: it will be dropped by the site routers. Thus we avoid making the designated router a central authority.

8.6 Caching in WAVEMQ

Generating and validating WAVE proofs, while reasonably fast, is still much slower than the time it takes to route and deliver a message. On the site router, significant time is spent on both generation and validation of a proof. On the designated router, only validation is performed.

To optimize proof generation, we can add a cache. The key in this cache is the concatenation of an entity private key and the desired policy, and the cache stores the resulting WAVE proof. This will allow proof generation to be skipped most of the time, only needing to occur if the previous proof becomes invalid due to expiry or revocation. For proof validation, we can cache the policy resulting from a validation, indexed by the hash of the proof, allowing us to skip the validation step when identical proofs are used for identical operations (as typically happens for a stream of published messages). In both cases, we make sure to invalidate the cache when there is an expiry and we check for revocations frequently (every few minutes). An administrator configures a maximum time that revocation checks can be skipped for in the event of the site losing internet connectivity, as revocation checks must contact the global storage tier.

In addition to caching the results of proof build and validation operations, we cache the proofs themselves so that they are not transmitted over the internet repeatedly. A proof can be hundreds of kilobytes, whereas a message payload may only be handful of bytes. Transmitting the proof with each message can drastically increase the bandwidth required for a message stream. Instead, when transmitting messages, the routers send the full proof on the first message and send the hash for each subsequent message. If the proof is not found in the cache on the remote side, it sends back a "send full proof" error and the originating router will retransmit the message with the full proof attached.

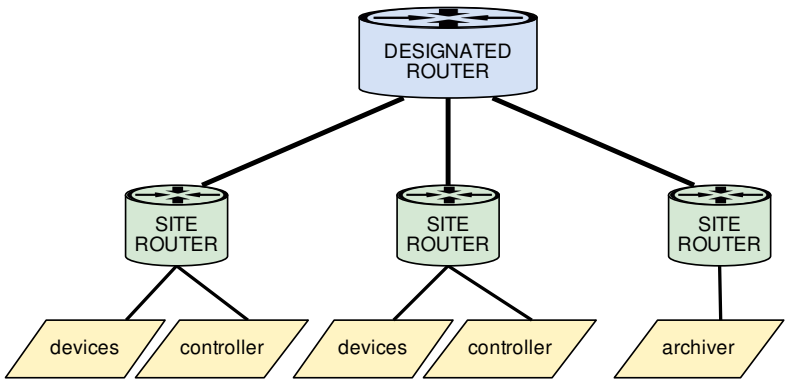
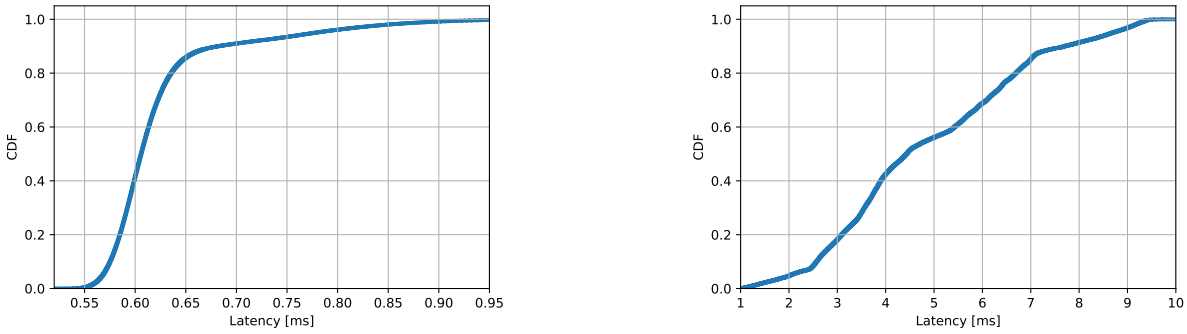


Figure 8.5: The WAVEMQ configuration used for latency benchmarking



(a) Latency through site router

(b) Latency through designated router

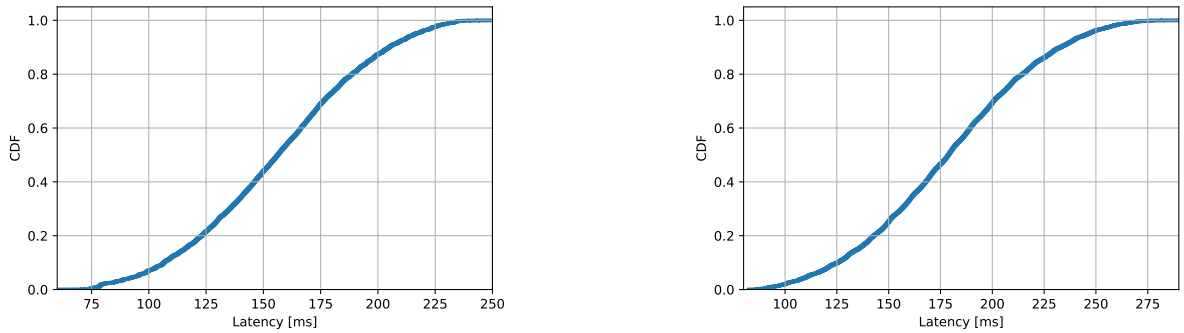
Figure 8.6: End-to-end latency CDF for 200 devices reporting at 1Hz with caching enabled

8.7 WAVEMQ Performance Evaluation

The most important performance metric for a syndication system is end-to-end latency. In particular, as the latency in WAVEMQ is the result of "busy" crypto work, the latency drives the throughput. The throughput of a router is roughly NumCores/Latency.

We configure three site routers and a designated router as illustrated in Fig. 8.5. These are emulated on AWS EC2 using an m5.4xlarge machines for each site, and one for the designated router. Each of the sites has a set of devices (either 100 or 10, depending on the experiment) and a controller that locally subscribes to all of the data from the devices within that site. The third site router has an archiver service which subscribes to all of the devices in both sites. We measure the latency from the device to the controller (site-local delivery) as well as from the device to the archiver (remote delivery).

The first experiment is with a normal caching configuration. Here proof generation and validation are cached. Fig. 8.6a shows the cumulative distribution function for the latency in the site-local delivery case where 95% of messages are delivered in under 0.75 ms. Fig. 8.6b shows



(a) Latency through site router

(b) Latency through designated router

Figure 8.7: End-to-end latency CDF for 20 devices reporting at 1Hz with no caching enabled

the case where the messages are delivered remotely, passing through the designated router in addition to two site routers. Here, 95% of messages are delivered in under 9ms. This performance is comparable to traditional publish/subscribe messaging systems. [97] evaluates several MQTT implementations and finds the end-to-end latency of message delivery under similar loads to be between 1 ms and 9 ms in the single-broker case.

To characterize the importance of caches, we run the same experiment with all caching in WAVEMQ disabled. Fig. 8.7a illustrates the latency between the send-message command on a device and when it is received on the controller. This latency is quite severe, with a mean of 160 ms, as every message must first update the perspective graph, build a proof and validate it. Fig. 8.7b has a mean of 180 ms as it must validate each proof an additional time in addition to the queuing overhead of transmission through the designated router.

This experiment highlights how important caching is for WAVE: the cached version of WAVEMQ has performance comparable to a traditional publish/subscribe system whereas the version with no caching has a very high latency that would reduce throughput and potentially be noticeable at an application level.

8.8 Hiding Proofs From The Router

In the XBOS model, the sites are independent trust domains and the traffic all flows to a single WAVEMQ designated router. This means that the designated router should not be trusted to enforce security policies or keep data private.

In the design of WAVEMQ presented thus far, the designated router is not entrusted with enforcing authorization policies - the site router validates all the proofs, so a compromised designated router cannot make a site router deliver invalid messages to clients. In addition, messages can be end-to-end encrypted, preventing the designated router from reading the contents of the messages. Nevertheless, the designated router is in a position to read the contents of the proofs as it requires this information to filter out invalid messages, giving us the denial of service resistance property.

The proofs show the chains of delegation which may reveal some organizational metadata. We can, however, alter the design of WAVEMQ slightly to remove the need for the router to see the contents of proofs. This section details two methods that can achieve this.

The first method uses anonymous proofs, as discussed in §5.10. The second leverages Intel Software Guard eXtensions (SGX).

Anonymous Proof

WAVE's anonymous proof scheme allows you to form a proof that you have permissions corresponding to a particular partition. Recall that a partition is a function of the policy and is more generic (there are multiple possible policies that map to the same partition).

This means that an anonymous proof may show that the prover has more permissions than they really do, but it will never show that they have less permissions. This means that it is safe for the designated router to drop messages from site routers that are unable to provide an anonymous proof showing permissions greater than those required to publish the given message (or begin the given subscription). There may be false positives – messages that get accepted when they should not be – but there will not be false negatives – messages that get dropped when they should not be. As an entity cannot generate an anonymous proof unless it has very similar permissions to what it is claiming, we still limit the scope of denial of service attacks as an attacker must have gained very similar permissions on a resource in the same namespace in order to mount an attack.

In this scheme the designated router learns the resource that the message must be sent on, as this is in plaintext, but it does not learn the information in the proof, such as which entities gave the sender permissions.

Using SGX

Proof verification could be moved inside an SGX enclave, which would allow full proofs (not anonymous proofs) to be verified without the designated router learning the contents of the proof. This would remove the possibility for false positives that exist in the anonymous proof mechanism above.

SGX, however, is all built on the assumption that the hardware is constructed correctly and has no flaws. As the technology is quite new, researchers are still discovering the limitations of SGX and discovering vulnerabilities in SGX programs and the SGX implementation [112, 69]. WAVE was constructed to rely upon cryptography, not trusted hardware.

If one is willing to use SGX, however, and accept the associated cost of trusting the hardware, then it makes sense to move the entire designated router inside the enclave. In this scenario, the operator of the designated router does not even learn the resource that the message is published on. The transport layer security would encrypt a channel from the site router (which is trusted) to the enclave inside the designated router.

If SGX were to be compromised, this would let the designated router see the content of the proofs (the resources and the entities that granted permissions) but it would not compromise the

security offered by WAVE: the designated router is still unable to forge or alter permissions, or forge messages.

We do not consider the SGX scenario further, as being willing to use SGX is a significant concession. [43] explores this technique in depth. If we were operating in that paradigm, the entire design of WAVE could be drastically simplified, reducing to a design that essentially mirrors that of traditional systems designed with a central authority (e.g. a central ACL contained within an enclave). In this case the central authority would be SGX, and transitively, Intel. WAVE is designed to avoid such central authorities.

8.9 Summary

Syndication is an ideal place to enforce authorization policy. It is abstract enough that different devices with a similar purpose can have a uniform policy applied to them, but syndication is also low enough level that reasonably complex authorization policies can be expressed. This is especially true if the resources exposed to the syndication layer are engineered for this purpose, as in §8.1.

WAVE's model of authorization works well in this paradigm, as validated by the use of WAVEMQ in our building deployments of XBOS (Chapter 7). With appropriate caching, performance is comparable to traditional publish subscribe systems whilst adding delegable authorization that is verifiable at all points in the system. This property allows for the compromise of a site router to remain isolated to that site, unlike in MQTT (§10.5). By additionally using techniques such as anonymous proofs, we can further reduce the trust in the designated router so that, if compromised, it is unable to view the permission structure normally found in proofs.

Chapter 9

Concerns and Implications

The development of WAVE has been driven by functional requirements established during our deployments in real buildings, but it also includes characteristics considered to be "objectively good" by the security community, such as privacy preservation and decentralization. In reality, these characteristics are not without costs. The lack of a central authority and the reduction in permission visibility have implications for the "good guys", not just attackers. This chapter discusses a few of these issues.

9.1 Auditability

At the start of this work, in §1.3, we discussed that if the owner of a building delegated permissions to a tenant and they further delegated within their company, that the building owner should not be able to see the permissions that the tenant delegated. This example is uncontentious because it crosses administrative domains, so we can readily agree that permission information should not cross this administrative boundary. What might be less obvious is that this principle applies to all delegations. Within the company, if the CEO delegates permission to a team lead, and the team lead further delegates to other employees, the CEO does not have any visibility into those delegations. This is academically "pure" because it avoids creating unnecessary authorities, at any level, but it is a fairly different pattern from the authorization systems that people are accustomed to.

With a traditional system, it is usually possible to look at the records stored on the central server and obtain a complete picture of who has which permissions. In WAVE, there is no single entity with the ability to view all the permissions that other entitites have: each entity can only discover attestations that relate to their own permissions.

This functionality is by design. It would be quite simple to allow the namespace entity to view all the permissions granted within the namespace if that property is desired. For example, upon attestation creation, the prover keys could be encrypted under the namespace entity's key and, when published to storage, the attestation hash is additionally added to the namespace entity's notification queue. If deterministic encryption is used, other entities can verify this procedure was

carried out correctly. We did not develop this mechanism because it goes against the design goals of the system. We are, however, concerned that perhaps these design goals, whilst academically desirable, might cause societal problems in practice.

For example, imagine an employee in a company decides to give his friend access to company resources. This is a perfectly legitimate delegation at a technical level: the employee is trusted with the permission to delegate. Unfortunately, they are abusing that trust, and the design of WAVE protects his privacy, therefore preventing, or at least frustrating, detection. If anonymous proofs (§5.10) are used, then even when using the service, the friend would not be revealing that they are not an employee.

There are technical solutions to this specific problem: perhaps in addition to proving permission, an entity needs to prove some identity claims (§3.7) such as `company/employee` which are granted without delegation, but this highlights that it is possible to construct a scenario where WAVE's privacy protection is problematic. Analagous issues arise in other settings where auditability is typically assumed.

Unfortunately this issue is still not simple. It is easy to construct a similar scenario that serves as a counter-example: imagine this was a customer of a cloud word processing suite and he delegated edit access to a friend: it is desirable that the service provider cannot distinguish between the original customer and the delegated-to friend and that the service provider remains completely unaware of the delegation. The delegations and resources of both of these scenarios could be identical, but the context makes one scenario "bad" and the other "good".

We do not have an answer to this family of problems. There are technical solutions that can be developed for each particular scenario, but they all have distinct trade-offs. We are not aware of a universally applicable change to WAVE that would remedy even a portion of these problems. Therefore, we advise that those people using these patterns of authorization think carefully about the implications of the system's functionality in their specific contexts.

9.2 Cryptographic Handoff

WAVE as a system differs from traditional authorization systems that solve roughly the same problem in that it enforces the majority of system properties using cryptography. In considering the implications of this choice, we can leverage a model created for this purpose: the Handoff model [90]. The handoff model considers the technological artifact as an assemblage of systems, or *components*, where each component could be a piece of software or hardware, but could also be a human. Each component affords a set of *modes* of interaction. For example, a light switch allows another component (the human) to switch it on or off (the mode).

The benefit of considering a technological artifact in this manner is that many advances in the technology can be expressed as a *handoff* from one component to another component that is replacing it. Whilst the component may serve an identical role (for example WAVE solves the same problems as OAuth, which it replaces), the handoff model elucidates the implications of the change in afforded modes of interaction.

Further, representing a system in the handoff model brings forward the changes to the *boundaries* of the system. As one replaces a component with another, the set of other components (technological and human) able to interact with the system, or their modes of interaction, may change. For example, where the government may previously have been able to interact with authorization data by issuing a subpoena to a service provider, under the WAVE system this mode of interaction is no longer afforded. Thus, we have moved the government beyond the boundary of the system, at least for this contextualization.

Finally, the handoff model surfaces the *triggers* that cause the handoff to occur. In our case, the triggers are described earlier in the dissertation, where we present the motivation for this work. Briefly, the trigger is a desire to avoid a central component and the modes of interaction (viewing or changing authorization policy) associated with it. It is interesting that the problems that surface in our handoff are directly as a result of solving the "problems" that triggered the handoff in the first place, not as a side effect of the particular *method* of solving the problem. We could analyze some of the implications of the handoff without the system even existing, by assuming that any given change to the system solves the triggering problem, and examining the consequences.

While there are many systems that this work affects (anything that uses authorization), we can omit the details beyond the boundary of the authorization system, as WAVE has minimal impact on the components beyond that boundary, at least when used purely for authorization (we consider more advanced use below). With some simplification, we can view a traditional centralized authorization system as consisting of four components: the user, the service provider, the authorization server and the authorization policy. Although there are other important components within the boundary of the system such as law enforcement or attackers, they interact with components in the system through the same modes afforded to the service provider, so we represent them all as the service provider for simplicity. In this system, the authorization policy is fully controlled by the authorization server, and the authorization server is operated by the service provider.

The user interacts with the authorization policy *through* the service provider. Thus, the service provider has complete control over which modes of interaction with the authorization policy the user is afforded. This is one of the triggers that caused us to pursue WAVE. For example, Google Docs does not let a user grant write access to another user without simultaneously granting the ability to share the document with others. Whilst this is something a user may very well want to do, it is not permitted by the service provider. The service provider, being the operator of the authorization server, has complete control over the authorization policy. When considering what is technically possible, there are essentially no restrictions of the actions that the service provider may take on the policy. In practice, there are non-technical regulations that restrict what the service provider is allowed to do. These are *normative* in that they control the modes of interaction by stating what *should* happen as opposed to what *can* happen. As an example, European companies under the GDPR are heavily fined for using user data in ways that the user did not consent to. This compels the company to treat user data in particular way, but does not *technically* limit them from, say, selling it for profit. We can model these regulatory forces as components in the system that, while not directly limiting the modes of interaction that the company has with user data, incentivizes certain modes over others.

Our key objection with traditional centralized system thus becomes clear: the service provider

may view and change the authorization policy arbitrarily if they are not suitably compelled by the normative regulations on that mode of interaction. Recall that we consider other key components in the system, such as attackers, as being synonymous with the service provider. Attackers likely believe they are not going to be caught, thus are likely not concerned with the legal implications of exercising the service provider's ability to view and modify the authorization policy. Thus the existence of this mode of interaction, and the lack of technical, non-normative restrictions on its use are problematic.

WAVE addresses this problem by removing this mode of interaction entirely. WAVE can be formulated in the handoff model as four components: the user, the service provider, the authorization policy and the cryptographic protocols. The modes of interaction between these four components differ substantially from those in the traditional model above. The service provider is no longer the operator of an authorization server. Instead both the service provider and the user interact with the authorization policy through the cryptographic protocols. The handoff here is the replacement of an authorization server with cryptographic protocols which, unlike the traditional model, affords the user more powerful modes of interaction than the service provider. The user is capable of viewing and altering the authorization policy (by granting and revoking permissions), whilst the service provider is only capable of viewing the pieces of the authorization policy that the user explicitly reveals to the service provider. We omit the ULDM storage server from the formulation of the components as the modes of interaction with the authorization policy it affords are entirely subsumed by the cryptographic protocols.

Given this formulation, we can observe that a key mode of interaction has been lost during the handoff from authorization server to cryptographic protocols: the service provider and other components acting through it such as law enforcement agencies, are no longer afforded the ability to observe the authorization policy. In the traditional model, this mode of interaction existed, but its use was moderated by various normative factors including legal regulation and social norms (fear of public outcry). By design, WAVE completely removes this mode of interaction. Where we previously had a mode of interaction with multiple "knobs" that could be tweaked to compensate for technological and societal changes over time, we now have a fairly static system that encodes a *snapshot* of what the engineers considered objectively good, at a single point in time. There is very little capacity for evolution in such a paradigm. The cryptographic protocols introduce a non-normative regulation of the modes of interaction that exist with the authorization policy. Instead of being something open to interpretation, or capable of being circumvented, it is now something concrete that affords no exceptions. This has some significant implications. For example, there may arise a scenario where it is in everyone's best interests for a service provider to reveal a portion of the authorization graph, but this action is simply not afforded by the cryptographic protocols, and cannot be done.

To better understand the implications of this handoff, it is worth considering a use case.

9.3 Liberty

We have presented an authorization system that does not let a service operator view who has permissions or how they have been delegated. Proof of this permission can be shown anonymously (§5.10), without revealing identity or how the permissions was obtained. We have also presented an end-to-end encryption mechanism that allows content to be encrypted under the authorization policy(§5.11), preventing the service operator from viewing the content. Collaborators on this work have developed a system that allows service providers to host anonymous cloud storage that hides who is accessing which file [79].

These primitives can be assembled into something resembling a fully-private Facebook. Users would each have their own namespace, and posts would be resources within that namespace. Posts would be encrypted under those resources before uploading to cloud storage. To share posts, users create WAVE attestations. This system would allow users to discover and share content in a manner similar to Facebook but with a completely different privacy model. In regular Facebook, the service operator can see everyone's posts, photos and social graphs and there are far too many examples where this power has been abused [114, 109]. In our version, the assemblage of our various techniques would keep the service operator completely oblivious, which intrinsically prevents them from mining personal data or selling it for profit. All aspects of function are regulated through non-normative technological means. Users themselves can determine who can do what with their content, rather than having to trust the service provider to enact that policy.

To extend the handoff model formulation above, we have now expanded the boundary of the system to include the components that provide the core functionality of the system – such as storing posts, sharing posts etc – and are using the cryptographic protocols to remove the modes of interaction that the service provider used to have with the user's content.

This system obviously has merit. It is a travesty that society has accepted the model of exchanging one's privacy in exchange for free digital services. This system breaks out of that paradigm, offering an alternative. Data mining on social platforms is credited with everything from enabling manipulative advertising to shifting the course of entire elections [75]. Allowing people to engage in online social activities without these repercussions could have profound positive societal implications.

At the same time, there are countless examples of where platforms like Twitter and Facebook used this power to censor hate speech, locate human trafficking rings, prevent terrorist attacks, etc. This is often manual, for example Facebook's banning of white supremacist organizations was done after a manual review, but there is a move towards automation of this. For example, Twitter experimented with a bots that detect users promulgating hate speech and automatically banned them. The experiment was terminated because it matched too many high-profile politicians, but after refinement may be re-introduced. These actions, manual or automatic, are not possible in our system. In our private social networking platform, terrorists and human traffickers are afforded the same right to privacy as everyone else, and this right is cryptographically protected. The ability to "make an exception" to save lives is no longer assumed. It is not clear which paradigm is better: everyone has privacy, including criminals, or nobody has privacy. Benjamin Franklin is oft quoted as saying "Those who would give up essential liberty, to purchase a little temporary safety, deserve

neither liberty nor safety". We have collectively given up a great deal of liberty, in the form of privacy, in our acceptance of these free online services. There are those that argue that this has bought us safety, but there are also those that argue that it mostly appears to have enabled abuse of our data for others' financial gain.

The danger of cryptographically enforcing a property such as privacy is that it becomes a categorical, irreversible declaration. While many of us would readily agree to the value of privacy, are we as ready to state that privacy must be upheld in absolutely every scenario? Would you feel the same way if it was a daughter who was kidnapped after agreeing to meet an online "friend" and you are told that the privacy of this "friend" is unbreakable?

It is worth noting that these issues arise chiefly when there is an interaction between the innocent parties, and malevolent parties. Often there is an argument that a new privacy-ensuring platform would enable criminals to communicate with no repercussions. This argument is fallacious: the technology for private illicit communication has existed for decades exemplified by platforms on the dark web such as Silk Road and there does not appear to be any way to restrict such activity. The debate here is around the consequences of pushing this technology out to everyone. What recourse exists when there are interactions between innocent parties and malevolent parties in a neutral forum, i.e. one whose use is not a tacit opt-in to criminal activity? When a child is stalked on social media, for example, can we locate the culprit? The distinction is that there is a party using a commonplace technological artifact with some implicit assumptions about safety, and these assumptions may prove to be incorrect. In the dark web, all parties involved are presumably aware of the consequences of their actions.

We do not have an answer to these questions, but answers need to be found. Engineering such a platform is absolutely possible given the technologies presented here and in recent work such as [79]. Given the complexities of the issue, we should begin considering the full implications of this now, lest the technology becomes readily available without due consideration.

9.4 Summary

The widespread violation of user privacy in online systems, coupled with the lack of reasonable security mechanisms in the IoT devices that permeate our lives have led us to conclude that the current set of regulatory forces are insufficient. This system removes the mode of interaction with authorization policy that was regulated through normative moral and legal means and replaces it with non-normative cryptographic regulation. This technology seems more than able to provide the privacy and security properties we set out to achieve, but we find ourselves unable to state that these properties are categorically desirable in every circumstance. Thus we are in a moral quandary: it is not clear which paradigm is truly better. We don't offer an answer, but hope that future work will investigate these aspects of the problem further.

Chapter 10

Related Work

Chapter 2 discusses related work that is core to the understanding of the work in this dissertation. This chapter extends on that, discussing aspects that serve to draw out nuances and comparisons, with our work firmly established.

10.1 Graph Based Authorization

As discussed in §2.3, our Graph Based Authorization (GBA) model is an extension of that found in SDSI/SPKI. This model of authorization has not seen a large amount of attention over the years, but it is worth discussing one notable example in more detail. Macaroons [20] is a 2014 NSDI paper that has many of the properties that we aim to provide. It is a delegable authorization scheme that shows proof of authorization as a chain of delegations. As a result, their delegations have the same autonomy property we aim for: a party can delegate without involving an authority in the process.

Delegation in Macaroons manifests as taking a "Macaroon" cookie that you possess granting you some permissions, appending some caveats to it to narrow the permissions you wish to re-delegate, and then transmitting the narrowed cookie to the party receiving permissions. To use the delegated permissions, the attenuated cookie is presented to the service provider, which can verify that none of the caveats in the cookie have been tampered with and that it is derived from a cookie initially issued by the service provider. After this, the policy encoded in the cookie is then verified, which we discuss later. It is not a sequence of objects being presented (like a proof being a path of attestations, in WAVE) but rather one object that is modified with each delegation in a verifiable manner.

This highlights the differences between WAVE and Macaroons, both in their goals and in the functionality they realize. Macaroons is built on the assumption that a person delegating permissions to a friend is in communication with that friend. They do not assume that channel is secure, but they do assume that the credential object can be transferred over it. In addition, they are only concerned with delegating permissions that one already has. The combination of these two properties allows Macaroons to operate without a discovery mechanism or global storage. The analog

in WAVE would be to forgo RDE and simply include a list of all required attestations when you grant the friend permissions, encrypted under their public key. This works, of course, but it prevents out-of-order grants and requires participants to be on-line. Macaroons is designed for very short-lived permissions, akin to the bearer-tokens it aims to replace, so the lack of out-of-order grants is tolerable. WAVE is designed for long term (e.g years) permissions so the requirement for out-of-order grants is critical to deal with key churn over time.

The other key difference is that Macaroons is service-provider-centric. Their context is one where some cloud provider initially gave parties permissions, which are then re-delegated. The only participant that ever needs to *verify* these delegated permissions is the initial service provider. This is the same context that SDSI/SPKI is designed for. In both of these systems, only the service provider can verify permissions. WAVE allows for the authority that initially granted permissions to be distinct from the parties that verify delegated permissions. For example, the private Facebook proposed in §9.3 would have the individual users be the authorities for their own namespaces, containing their posts. The service provider needs to validate proofs, but they are not the authority. Neither the SDSI/SPKI nor Macaroons model would work here.

In exchange for these limitations, Macaroons is a simpler system and has higher performance. Thus, we consider Macaroons and WAVE to be complementary: Macaroons is the better solution for their context, and WAVE for our context.

Distinct from the mechanisms for delegation, Macaroons also has an interesting policy mechanism. We discuss this separately below.

10.2 Authorization Languages

WAVE focuses on how permissions are delegated and discovered while maintaining privacy. This does not require a specific mechanism for describing authorization policy, only that it is possible to instantiate a policy-aware RDE by defining partition mapping functions (§5.3). As we are using the system in deployments, we found it necessary to complete the system by choosing a particular policy language (RTree, §3.3) but we do not regard RTree as a contribution of this work.

Authorization languages, with many formally-proven properties, have been developed over the course of the past few decades [20, 95, 21, 84, 14, 53]. An advanced authorization language allows one to express a policy with complex predicates. As an example, a predicate may ensure that an actuation command may only be executed if it results in a "safe" configuration or if it occurs at a particular time of day. As discussed in §7.3 we have found that such requirements can be met by enforcing them within the service itself, but it is possible to have a versatile authorization policy that can express these constraints. The advantage of expressing these all within the policy is that one can process the policy to obtain formal guarantees about reachable states.

The *third party caveat* authorization mechanism in Macaroons [20] adds a richness to delegation mechanics that would be valuable in future versions of WAVE. With Macaroons' caveats, one can add arbitrary predicates to delegations, as above, but one can also require the presence of another authorization proof. For example, a manager could grant permission to set a thermostat setpoint to an employee, but specify that the permission is only valid if the employee can concur-

rently provide a proof from an occupancy service showing that the employee is physically within the room they are trying to configure. This functionality is extremely useful. Consider the example given in §9.1 where an employee grants access on company resources to his friend. We proposed that the solution to this problem might be to have the service require an identity claim proving that all parties are employees. In Macaroons, this can be expressed directly in the authorization policy: when the initial permission is given to the employee, it can be restricted with a third-party caveat requiring that a "discharging service" responsible for identifying employees has confirmed that the user is an employee. This caveat would stay with the authorization so that any delegation to a non-employee is rendered invalid. While many arbitrary types of policy can be represented in RTree through careful encoding of resources and permissions, this third-party caveat and discharge mechanism can not. It would be a reasonable extension to WAVE to allow RTree statements to be predicated upon concurrent proving of other RTree statements.

10.3 Hidden Credentials

Hidden Credentials [113, 73, 58, 89], a subset of Trust Management, aims to protect the *required policy* as well as avoiding revealing that a party is in possession of specific credentials. This is best understood by example. Say there is a server S and a client C. The C wishes to access S so it asks "what permissions do I need to access you". Typically, S would just respond with what is required. In Hidden Credentials, they consider this required policy to be sensitive information, so they propose a protocol that allows S to validate C's permissions without revealing what the desired permissions are. In addition, if C is accidentally accessing an imposter server K, then C does not wish to reveal to K that they have permission to access S, so C will require some proof that K is valid before sending their credentials, whilst also not betraying any information about the nature of S such as the policy that C would consider sufficient to prove that K is S. Thus, Hidden Credentials allows both sides to slowly build up trust in each other without revealing too much in advance.

WAVE does not attempt to solve this problem. In WAVE we assume that C would know in advance what permissions it requires to access S (e.g. if you are writing to `device/resource` then it is easy to know a priori that you require `app::write` on `device/resource`). In addition, we do not consider a proof that S is valid to be sensitive information. For example, in WAVEMQ, when a client connects, the router sends a proof that the router is authorized to route messages on the requested namespace. In Hidden Credentials this would be considered unacceptable.

We believe that WAVE's position on this is fair. In mainstream systems, the identity of a service provider is not usually a secret, nor is the required policy. If one accesses a website, it will readily provide a TLS certificate proving its identity. Similarly if you attempt to access a protected resource such as a private document on Google Docs, it will readily tell you that you do not have permission, and even provide a button allowing you to request the permissions you need. Thus, WAVE's position is a common one. We have identified aspects of existing systems that be believe are important to fix (§2.1), and we have done so, but there are still problems identified as important by authorization literature that we do not solve.

10.4 Storage

The Map Log Root used in WAVE's ULDM is similar to the approach used by CONIKS [88] and Key Transparency (KT) [64] in that it ensures the integrity of the main Merkle map. There are, however, several differences between a ULDM and the CONIKS/KT data structures. As a ULDM does not need to prevent iteration of the contents, it can be log derived, allowing an efficient verification that it is append-only. Specifically, the map in a ULDM is constructed by applying the operation log, so verification can be done by replaying the operation log and comparing the outcomes. In contrast, CONIKS/KT requires every user to check every epoch of the map to ensure the values stored match expectations. This approach would not work for our use case as we expect every user to create hundreds or thousands of objects, and requiring every user to check each of these objects at every map epoch is intractable. The ULDM approach 1) reduces the amount of work as it scales with the number of *additions* to the map rather than the *size* of the map, as in CONIKS, and 2) places the majority of the burden on auditors, rather than users who may be offline.

Revocation Transparency [81] is also similar to a ULDM. It was posted as an informal short note, and to our knowledge, it was never fully developed. It lacks the Operation Log, so one cannot validate the append-only nature of the map by replaying the log. Instead, an implementation of RT would require the client/auditor to request a consistency proof between two versions of the map without knowing the contents. While this is technically possible, we are not aware of any Merkle tree map databases that support this operation. Our approach in the ULDM is simpler and is built using operations provided by an off-the-shelf database, such as Trillian [67], with full auditability.

10.5 Publish/Subscribe

WAVEMQ, presented in Chapter 8, is a tiered publish/subscribe communication service. This section briefly discusses how authorization and tiering are typically achieved using existing systems such as MQTT [13].

Tiering in MQTT is called *bridging* and works by configuring one router to forward messages to another router. This is manifest as a synthetic subscription on the source router to all messages matching the bridging pattern. When messages are delivered to that subscription they are published to the remote router as if they were being published by an ordinary client. This is logically similar to WAVEMQ, but there are two large differences. Firstly, in WAVEMQ, traffic is only "bridged" between sites if there is a subscriber on the remote side that has requested the data. This reduces the bandwidth used for connecting sites. Secondly, WAVEMQ maintains queues for individual subscriptions on the bridging link. MQTT does not do this, so when there is an interruption to the connection, a high frequency topic can fill the single queue causing message loss in all other topics.

The biggest differences between MQTT and WAVEMQ are with respect to authorization. The MQTT authorization model has clients authenticate with a broker, to establish a session. When a client publishes or subscribes to a resource in that session, the action is compared against an

ACL stored on the broker to determine if the action is legitimate. If it is approved, the message is published, but there are no further validations of authorization on that message. The bridging mechanism will pick up on the published message and forward it to the remote broker, but the message does not contain any record of who published it. As a result, the remote broker has no choice but to treat all incoming bridged messages as authorized. I.e. authorization is established once at the message's origin and never checked again.

In this model, if one broker is compromised, the entire network of bridged brokers is compromised as arbitrary traffic can be injected on any topic and every message is treated as trusted. This "compromise" may also result from misconfiguration as every broker maintains its own ACL for users. If a user is accidentally configured with too many permissions at one broker, it can use that broker to send messages to any client connected to any bridged router, even if the sender does not have those permissions in any other ACL.

One of the principles of WAVE is that each message should carry a proof of authorization. This allows the message to be validated at multiple points, including validation by the final message recipient. This is particularly important if the message is flowing across administrative domains or if one wishes to isolate the compromise of one system from another. If a WAVEMQ site router is compromised, it would only affect the one site as it would be unable to generate the proofs to support arbitrary messages for other sites.

Chapter 11

Conclusion

11.1 Summary of Work

Almost all computer systems, from social media sites to movie streaming services, from smart thermostats to nuclear power station reactor controllers, involve an authorization subsystem. This subsystem has the final word on which actions are allowed, and by whom. At present, the architectures of these critical subsystems are predominantly centralized, despite providing authorization for a system that may be distributed.

A centralized system is more vulnerable to attack, and the compromise of an authorization system cascades throughout all systems that depend on it. Even during normal operation, a centralized system poses problems; the operator has full visibility into the permission data of all users which may reveal private information, such as social relations or the structure of internal processes.

In addition to having an inherently insecure construction, widely used authorization systems have poor support for delegation which complicates the management of permissions; the lack of transitivity means that resource owners cannot reason about how or why a user has permissions, and revocation of a user has no effect on anyone they may have delegated to. In some systems, delegation is entirely absent which forces users to share their identity (i.e. username/password) instead of sharing permissions. This does not allow for any attenuation, giving the granted user full control, including aspects of the service, such as billing and the ability to delete the account, which is clearly unacceptable.

This work set out to address these problems by answering the question:

> How can we construct an authorization system that supports first-class transitive delegation across administrative domains without trusting a central authority or compromising on privacy?

To ensure the result is practical and valuable beyond academic discourse, we ground the solution in a specific use case and further answer the question:

> How can we leverage such a system to improve the security of IoT communications?

We begin answering these questions by identifying that a Graph Based Authorization (GBA) model is capable of representing transitive delegation as a first class primitive (Chapter 3). Traditional central-ACL systems as used in Google Docs or Dropbox do not store the provenance of a user's permission, making it impossible to predicate a user's permissions on the user that granted them permissions, required for transitivity. GBA represents permissions as a graph of the actual delegations, and represents a proof of authorization as a path through the graph. Thus, GBA inherently captures the provenance of permission, and offers transitive delegation. If one of those delegations were revoked, for example, this would be observable as a missing link in the proof of authorization, invalidating it.

While GBA provides transitive delegation, it does not intrinsically permit this across administrative domains without trusting a central authority. As an example, SDSI/SPKI [95] provides GBA but retains a central authority. We identify that this problem can be solved by constructing a global storage tier that proves trustworthiness cryptographically, and placing the graph in this storage.

We present the Unequivocable Log Derived Map, a storage data structure that leverages consistency proofs over Merkle trees to ensure that the integrity of the storage is maintained, even if the operator is actively malicious. This allows for horizontally scalable global storage (unlike a blockchain), whilst still not introducing any central authorities, as the storage servers have no choice but to act honestly.

The global, fully open nature of the ULDM storage tier forces us to consider the next aspect of the thesis question: can we do this without compromising on privacy. In isolation, the ULDM still allows the operator visibility into the permission data for all users, which is unacceptable. To resolve this, we present Reverse Discoverable Encryption (Chapter 5). This is a method for encrypting attestations such that they can only be read by parties that could use the permissions in a proof. RDE introduces a novel use of identity based encryption variants that allows the protocol to function without any communication between participants and permits permission grants to occur in any order.

To further evaluate our proposed answer comprising GBA, ULDM storage and RDE, we construct a full instantiation of the entire system, WAVE (Chapter 6) and evaluate its performance. We establish that our techniques are amenable to high performance implementation. Performance of common authorization operations is comparable to existing authorization systems, despite providing much stronger security guarantees. This affirms that the combination of GBA, ULDM and RDE is a practical answer to the first thesis question.

To answer the second thesis question, we propose a microservice architecture for Building Operating Systems and IoT in general (Chapter 7). This architecture is constructed around an information bus that enforces WAVE authorization and a set of driver services on that bus that abstract individual devices. Such an architecture allows for the management of permissions in the IoT setting to performed using homogeneous WAVE delegations, even though the underlying devices may offer heterogeneous native authorization interfaces. We further gain the benefits of GBA, such as intuitive revocation due to the transitivity of delegations.

An instantiation of this WAVE-enforcing information bus, WAVEMQ, is presented in Chapter 8. We find that by attaching WAVE proofs to each message and validating these proofs in the

core of the network, we obtain denial of service resistance. This is a valuable property for IoT devices that do not have the luxury of being over-provisioned which is the typical solution for DoS resistance in a cloud setting. We further provide techniques for enhancing the information bus so that it is resilient against intermittent connectivity. Finally, we develop caching techniques and evaluate the implemented system. We find that WAVEMQ exhibits low end-to-end latency, comparable to existing publish/subscribe systems, which affirms the practicality of WAVE in real systems.

The introduction of cryptographically-enforced privacy has both positive and negative societal implications. Chapter 9 discusses how activities such as auditing are impacted by this increased privacy in WAVE. We further consider the construction of a privacy-preserving social networking site, using WAVE, and the societal implications of such technology. In such a paradigm, criminals are as protected as the innocent, and the right to privacy is categorically inalienable. Many uncomfortable scenarios can be imagined, and it is hard to decide if this future is better or worse than the status quo. While we do not offer answers to the questions that arise, we identify points for future consideration.

11.2 Looking Forward

While this work has placed an emphasis on the context of smart buildings and IoT, these techniques are applicable in most scenarios where authorization is present, which is everywhere. Certainly, authorization systems like OAuth and Active Directory are omnipresent in a variety of fields. WAVE is capable of providing much of the functionality that these systems provide and with minor work could act as a drop-in replacement. Thus, we could bring cross-administrative-domain transitive delegation to a variety of systems such as those in finance, entertainment or communications. Systems that must be resilient to localized compromise, such as the electric grid, may stand to benefit the most from WAVE's removal of central authorities.

Given this widespread applicability, perhaps it would be worth constructing a single unified infrastructure. DNS began as a fragmented set of `/etc/hosts` files on machines across the internet, and evolved to become a single distributed, but unified, system; the standard for name resolution. Perhaps we could construct a similar distributed-but-unified system for authorization, which would have the advantage of allowing people to maintain a small number of digital identities rather than the hundreds of individual per-system identities required by the current approach. An identity would be valid across multiple administrative domains, and services could interact with each other's permissions seamlessly without the complex federation procedures required today.

With such a system in existence, service providers would not have to manage their own authorization servers and could instead use the unified but decentralized global authorization system. There are both benefits and costs to this, and it is worth considering this further.

Whilst WAVE answers the questions we set out to answer, several new questions became apparent as the system developed. These may be of interest to future work.

ULDM abuse: The ULDM is, by design, open to all. At present there is no mechanism for preventing abuse of the system, such as a botnet repeatedly storing attestations that serve no purpose.

This problem is shared by many open systems, such as DNS, and we envision that future work may investigate techniques for reducing the potential for abuse. Perhaps rate-limiting requests from individual IP addresses or adding some kind of proof-of-work token with each request.

ULDM side channels: While we provide privacy in the construction of attestation objects, there are many forms of metadata in the interactions with a ULDM server that may leak sensitive information. These include the network information (IP addresses), the timing of requests, and the size of the attestation ciphertexts. Future work may investigate the application of techniques commonly researched in anonymous file sharing systems [32].

RDE visibility: RDE, as constructed, relies on the notion of a partition based on a *resource prefix* to govern the visibility of an attestation. This is intuitive for users, but there may be other methods that reduce the number of attestations that are decryptable but unusable in a proof. Of particular interest is the use of KP-ABE to include the full resource instead of just a prefix in the attestation encryption key, as discussed in §5.7.

Object serialization: In the WAVE implementation, benchmarking indicated that a large portion of the CPU time is spent serializing objects before they are written to the local database. This is due to the use of the generic Gob [60] encoding format. A purpose-built format would likely drastically reduce the overhead of writing to the database and could obtain a 20-40% speed up on operations other than proof verification.

Auditability: As discussed in §9.1, it is possible to add a mechanism to WAVE that allows namespace entities to completely view the attestation graph within their namespace. Whilst this goes against the design goals of WAVE, it may be required in practice if adopters have become accustomed to this capability in traditional systems. Prior versions of WAVE had this capability, and we found it immensely useful in identifying users with too many or too few permissions, which may actually increase overall system security.

The system has been constructed as a prototype, and there would be additional work to convert this into a production system suitable for wide-scale use. In particular, a rigorous security audit of the code would be required before it could be fully trusted. In practice, it would be wise for WAVE to be implemented by several parties, with a common object format. This would likely help catch bugs in implementations and limit the impact of undiscovered bugs.

11.3 Final Thoughts

We are excited to find that constructing a decentralized, delegable authorization system that preserves privacy is not only possible but that it is practical. Observing that the latency of common authorization operations in this paradigm is comparable to existing systems is encouraging considering that cryptographic solutions to problems are often associated with unrealistically high computational costs. While our implementation may require some work before it is a trustworthy, production-ready system, we believe that the techniques presented in this work are a solid foundation upon which such a production system could be constructed.

We believe that future work would need to focus on the usability of the system, as the complexity is somewhat higher than traditional authorization systems. We suspect most of this complexity can be absorbed by appropriate tooling and user interface design. As this work progresses, we encourage consideration of the implications of the system, especially with respect to auditability and corner cases such as criminal use.

Appendix A

RDE Security Guarantee

Below, we develop definitions to precisely describe the global authorization graph, and then we use them to formalize RDE's security guarantee.

Definition 6 (Path)**.** *Let x and y be entities. $(A_1, \ldots, A_n)$ is a **path** from x to y if either $n > 0$ and $A_1.\mathsf{issuer} = x$, $A_n.\mathsf{subject} = y$, and $A_i.\mathsf{subject} = A_{i+1}.\mathsf{issuer}$ for all $i \in \{1, \ldots n-1\}$, or $n = 0$ and $x = y$.*

Definition 7 (Compatibility)**.** *Let A and B be attestations such that $A.\mathsf{subject} = B.\mathsf{issuer}$. We write $A \rightsquigarrow B$ and say "A is **partition-compatible** with B" if a key corresponding to one of the ID^*s in $Q(A.\mathsf{policy})$ can decrypt a WKD-IBE ciphertext with the ID $P(B.\mathsf{policy})$. We analogously write $A \rightarrowtail B$ and say "A is **partition-label-compatible** with B" if a key corresponding to one of the ID^*s in $M(A.\mathsf{policy})$ can decrypt an IBE ciphertext with the ID $L(B.\mathsf{policy})$. We extend this to paths as follows. A path $(A_1, \ldots, A_n)$ is **partition-compatible** if either $n = 0$, or $A_i \rightsquigarrow A_{i+1}$ for all $i \in \{1, \ldots, n-1\}$. A path $(A_1, \ldots, A_n)$ is **partition-label-compatible** if either $n = 0$, or $A_1 \rightarrowtail A_2$ and $(A_2, \ldots, A_n)$ is partition-compatible.*

Based on our definitions of P, Q, L, and M in §5.3 and §5.4, we can attach semantic meaning to compatibility:

Note 1 (Compatibility Semantics for RTree)**.** *$A \rightsquigarrow B$ means that $A.\mathsf{policy}$ and $B.\mathsf{policy}$ have overlapping time ranges, URIs with the same namespace, and the same permission string. $A \rightarrowtail B$ means that $A.\mathsf{policy}$ and $B.\mathsf{policy}$ have URIs with the same namespace.*

Now, we formally define the states attached to an attestations during the discovery process (§5.6) so we can later express the leakage of an attestation in each state.

Definition 8 (Attestation State Machine)**.** *Let A be an attestation. If there exists a partition-compatible path $p = (A, P_1, \ldots, P_n)$ to an entity compromised by Adv, then we say that A is **useful** with respect to Adv.*

*Otherwise, if there exists a partition-label-compatible path $p = (A, P_1, \ldots, P_n)$ to an entity compromised by Adv, then we say that A is **partition-known** with respect to Adv.*

Otherwise, if there exists a partition-compatible path from A.subject *to an entity compromised by* Adv*, then we say that* A *is* ***interesting*** *with respect to* Adv.

Otherwise, we say that A *is* ***unknown*** *with respect to* Adv.

From D's perspective in Fig. 5.1, for example, #1, #4, and #3 are useful, #5 is partition-known, and #2 is unknown. The components of an RTree policy are described in §3.3.

Based on Definition 8, we can now *informally* state the security guarantee of RDE. Let A be an attestation such that there does not exist a partition-compatible path from A.subject to a partition-compatible cycle in the global authorization graph. If A is **unknown** or **interesting** with respect to Adv, then Adv learns nothing about A except A.subject and A's revocation commitment. If A is **partition-known** with respect to Adv, then Adv learns nothing about A except (1) A.subject, and (2) $P(A.\text{policy})$. If A is useful with respect to Adv, then Adv can decrypt A and see all of its fields.

We now formalize the security guarantee of RDE as a game played by a challenger Chl and an adversary Adv.

Guarantee 2 (RDE)**.** *Let* λ *denote the security parameter. Consider any list of entities in the system, represented as names in* $\{0,1\}^*$*, any subset of these entities compromised by* Adv*, and any two authorization graphs* G_0 *and* G_1 *each described as a list of attestations in terms of the entity names, subject to the constraints below:*

1. $|G_0| = |G_1|$ *and attestations at position* i *in the lists of* G_0 *and* G_1 *must have the same length. We say that these two attestations* ***correspond****.*
2. *Corresponding attestations must have the same state* ***unknown/interesting/partition-known/useful*** *w.r.t.* Adv.
3. *If corresponding attestations are* ***useful*** *to* Adv*, or if either has a partition-compatible path from its subject to a partition-compatible cycle, then they must be identical.*
4. *If corresponding attestations* A_0 *and* A_1 *are* ***partition-known*** *to* Adv*, or if there exists a partition-label-compatible path from* A_0.subject *(or* A_1.subject*) to a partition-compatible cycle in* G_0 *(or* G_1*), they must have the same subject and revocation commitment and satisfy* $P(A_0) = P(A_1)$*, but may otherwise differ arbitrarily.*
5. *If corresponding attestations are* ***unknown*** *or* ***interesting*** *to* Adv *(and if there is no partition-label-compatible path from the subject to a partition-compatible cycle) then they must have the same subject and revocation commitment, but may otherwise differ arbitrarily.*

Each attestation in the graph is described in terms of the information in §3.3, not RDE ciphertexts. RDE guarantees that Adv*'s advantage in the following game is negligible in the security parameter* λ*:*

Initialization*.* Chl *generates each entity's keypairs. It sends to* Adv *the public keys (verification key and WKD-IBE/IBE public parameters) corresponding to each entity. For entities corresponding to malicious users,* Chl *also provides the secret keys (signing key and WKD-IBE/IBE master keys). Furthermore,* Chl *chooses a random bit* $b \in \{0,1\}$*, computes the RDE ciphertext for each attestation in* G_b*, and gives them to* Adv.

Guess*.* Adv *outputs a bit* $b' \in \{0,1\}$*. The adversary's advantage in the game is defined as* $\left|\Pr[b = b'] - \frac{1}{2}\right|$.

The constraints on cycles in Conditions #3, #4, and #5 are due to the lack of KDM-security for the WIBE and IBE used. It may be possible to remove these constraints with KDM-secure variants.

Proof Sketch for Guarantee 2. We define a new game in which Adv has no advantage and prove via a hybrid argument that Adv's advantage in the real game differs from its advantage in this new game by at most a negligible amount.

In the hybrid argument, each hybrid represents a game. In the sequence of hybrids, the encrypted graph provided by the challenger if $b = 0$ is identical to the encrypted graph in the previous hybrid, except that either (1) one of the WIBE or IBE ciphertexts generated by Chl in the Challenge phase is replaced with an encryption of a different string of correct length, or (2) the ID used for IBE encryption is changed to a different ID. Adv cannot distinguish between adjacent hybrids due to CPA-security of WIBE and IBE in case (1), and due to the *anonymity* of IBE in case (2). Because adjacent hybrids are indistinguishable to Adv, the difference in its advantage in adjacent hybrids is negligible. The first game is the real game (Guarantee 2). In the final game, Adv's advantage is 0. By the hybrid argument, we can conclude that Adv's advantage in the real game is negligible.

The order in which ciphertexts are replaced must be chosen carefully. This is because a ciphertext cannot be replaced with an encryption of zero if a secret key to decrypt the ciphertext exists in the graph. We now describe the hybrids.

We identify attestations in the graph in Conditions #4 and #5. Observe that the "partition-compatible" relation defines a directed graph over these attestations in each G_0 and G_1, where each attestation is a vertex and edges indicate partition-compatibility. We denote these new graphs S_0 and S_1. Both S_0 and S_1 are directed acyclic graphs, due to the stipulations in Conditions #4 and #5 regarding cycles. Thus, S_0 and S_1 can be linearized. Via a sequence of hybrids, we first replace ciphertexts provided by Chl when it chooses $b = 0$ with encryptions of a dummy "zero string," following the reverse order of S_0's linearization. For attestations in Condition #4, we replace the WKD-IBE ciphertexts in the attestations with encryptions of zero, in a single hybrid game for each attestation. For each attestation in Condition #5, we make two hybrid games; the first replaces its IBE ciphertext with an encryption of zeros, and the second replaces the ID used to encrypt with IBE for that ciphertext with a dummy ID. At the end of this hybrid sequence, the challenger provides a graph containing encryptions of zero in non-useful attestations if $b = 0$, and a proper encryption of G_1 if $b = 1$.

This is followed by another sequence of hybrids where we similarly transform the encryptions of zero provided by the challenger if $b = 0$ to proper encryptions of the attestations in G_1. This is done by transforming attestations in the forward order of S_1's linearization. In the final game, the challenger provides a graph containing a proper encryption of G_1, regardless of the chosen bit b, so Adv's advantage is 0. This completes the proof sketch. □

Discovering an Attestation

We revise the attestation discovery algorithm in §5.6 according to our generalization in this appendix.

1. The client adds edge A to the perspective subgraph.
2. The client searches its local index for F-type keys it has received via attestations from A.subject, and tries to decrypt A's outer layer using each key. If none of the keys work, it marks A as **interesting**, and it stops processing attestation A.
3. Having decrypted the outer layer in the previous step, the client can see A.partition. It searches its index for a E-type keys received via attestations from A.subject that are at least as general as A.partition. Unlike the previous step, this lookup is indexed. If the client does not have a suitable key, it marks A as **partition-known** and stops processing A.
4. Having completed the previous step, the client marks A as **useful** and can now see all fields in A. The client adds E-type and F-type keys delegated via A to its index, as keys in the systems of A.issuer.
5. If the vertex A.issuer is not part of the perspective subgraph, then the client adds it, and then requests the storage layer for all attestations whose subject is A.issuer. They are processed by recursively invoking this algorithm, starting at Step 1 above.
6. If A.issuer is already in the perspective subgraph:
   - For each F-type key included in A, the client searches its local index for **interesting** attestations whose subject is A.issuer, and processes them starting at Step 2 above.
   - For each E-type key, the client searches its local index for matching **partition-known** attestations whose subject is A.issuer, and processes them starting at Step 3 above.

This constitutes a depth-first traversal to discover and decrypt new parts of the authorization graph revealed by the attestation.

Bibliography

[1] M. Abdalla, E. Kiltz, and G. Neven. *Generalized Key Delegation for Hierarchical Identity-Based Encryption*. Cryptology ePrint Archive, Report 2007/221.

[2] Yuvraj Agarwal et al. "Buildingdepot: an extensible and distributed architecture for building data storage, access and sharing". In: *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*. ACM. 2012, pp. 64–71.

[3] Ava Ahadipour and Martin Schanzenbach. "A Survey on Authorization in Distributed Systems: Information Storage, Data Retrieval and Trust Evaluation". In: *IEEE Trustcom/BigDataSE/ICESS*. 2017, pp. 1016–1023.

[4] Bora Akyol et al. "Volttron: An agent execution platform for the electric power system". In: *Third international workshop on agent technologies for energy systems valencia, spain*. 2012.

[5] AllSeen Alliance. *AllJoyn: proximity based peer-to-peer technology*. `https://web.archive.org/web/20170205050111/https://allseenalliance.org/framework`. 2017.

[6] Michael P Andersen et al. "Democratizing authority in the built environment". In: *ACM Transactions on Sensor Networks (TOSN)* 14.3-4 (2018), p. 17.

[7] Michael P Andersen et al. "WAVE: A Decentralized Authorization Framework with Transitive Delegation". In: *USENIX Security*. 2019.

[8] Anonymized. *Open source code for WAVE*. Anonymized.

[9] Omid Ardakanian, Arka Bhattacharya, and David Culler. "Non-intrusive techniques for establishing occupancy related energy savings in commercial buildings". In: *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*. ACM. 2016, pp. 21–30.

[10] Pandarasamy Arjunan et al. "SensorAct: a privacy and security aware federated middleware for building management". In: *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*. ACM. 2012, pp. 80–87.

[11] Bharathan Balaji et al. "Brick: Towards a unified metadata schema for buildings". In: *Proceedings of the ACM International Conference on Embedded Systems for Energy-Efficient Built Environments (BuildSys). ACM*. 2016.

[12] Dana H Ballard et al. “Deictic codes for the embodiment of cognition”. In: *Behavioral and Brain Sciences* 20.4 (1997), pp. 723–742.

[13] Andrew Banks and Rahul Gupta. “MQTT Version 3.1. 1”. In: *OASIS standard* (2014).

[14] Moritz Y Becker et al. “SecPAL: Design and semantics of a decentralized authorization language”. In: *Journal of Computer Security* 18.4 (2010), pp. 619–665.

[15] David Belson. “Akamai state of the Internet connectivity report, Q4 2016”. In: (Nov. 2016).

[16] Daniel J Bernstein. “Curve25519: new Diffie-Hellman speed records”. In: *International Workshop on Public Key Cryptography*. Springer. 2006, pp. 207–228.

[17] Daniel J Bernstein et al. “High-speed high-security signatures”. In: *Journal of Cryptographic Engineering* 2.2 (2012), pp. 77–89.

[18] Elisa Bertino, Elena Ferrari, and Anna Squicciarini. “Trust negotiations: concepts, systems, and languages”. In: *Computing in science & engineering* 6.4 (2004).

[19] Betsy (Adrienne Elizabeth) Beyer et al. “Migrating to BeyondCorp: Maintaining Productivity While Improving Security”. In: *Login* Summer 2017, VOl 42, No 2 (2017). ISSN 1044-6397.

[20] Arnar Birgisson et al. “Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud.” In: *NDSS*. 2014.

[21] Matt Blaze et al. “KeyNote: Trust management for public-key infrastructures”. In: *International Workshop on Security Protocols*. 1998, pp. 59–63.

[22] Matt Blaze, Joan Feigenbaum, and Jack Lacy. “Decentralized trust management”. In: *IEEE S & P*. IEEE. 1996.

[23] Matt Blaze, Joan Feigenbaum, and Martin Strauss. “Compliance checking in the policymaker trust management system”. In: *International Conference on Financial Cryptography*. Springer. 1998, pp. 254–274.

[24] D. Boneh and X. Boyen. “Efficient Selective-ID Secure Identity-Based Encryption Without Random Oracles”. In: *EUROCRYPT*. 2004.

[25] Dan Boneh and Matthew Franklin. “Identity-Based Encryption from the Weil Pairing”. In: *SIAM J. of Computing,* 2003.

[26] S. Bowe. *BLS12-381: New zk-SNARK Elliptic Curve Construction*. `https://web.archive.org/web/20190530081114/https://z.cash/blog/new-snark-curve/`. 2018.

[27] M. Buevich et al. “Respawn: A Distributed Multi-resolution Time-Series Datastore”. In: *2013 IEEE 34th Real-Time Systems Symposium*. Dec. 2013, pp. 288–297. DOI: `10.1109/RTSS.2013.36`.

[28] Christian Cachin. “Architecture of the Hyperledger blockchain fabric”. In: 2016.

[29] Stuartk Card, THOMASP MORAN, and Allen Newell. "The model human processor- An engineering model of human performance". In: *Handbook of perception and human performance*. 2.45–1 (1986).

[30] Kaifei Chen et al. "Snaplink: Fast and accurate vision-based appliance control in large commercial buildings". In: *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 1.4 (2018), p. 129.

[31] Ke Chen, Kai Hwang, and Gang Chen. "Heuristic discovery of role-based trust chains in peer-to-peer networks". In: *IEEE TPDS* 20.1 (2009), pp. 83–96.

[32] Tom Chothia and Konstantinos Chatzikokolakis. "A survey of anonymous peer-to-peer file-sharing". In: *International Conference on Embedded and Ubiquitous Computing*. Springer. 2005, pp. 744–755.

[33] Dwaine Clarke et al. "Certificate chain discovery in SPKI/SDSI". In: *Journal of Computer Security* (2001).

[34] Community developed LDAP software. *OpenLDAP*. `https://www.openldap.org`.

[35] James C Corbett et al. "Spanner: Google's globally distributed database". In: *ACM TOCS* 31.3 (2013), p. 8.

[36] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. "Riposte: An anonymous messaging system handling millions of users". In: *Security and Privacy (SP), 2015 IEEE Symposium on*. IEEE. 2015, pp. 321–338.

[37] Kyle Croman et al. "On scaling decentralized blockchains". In: *International Conference on Financial Cryptography and Data Security*. Springer. 2016, pp. 106–125.

[38] Stephen Dawson-Haggerty et al. "BOSS: Building Operating System Services." In: *NSDI*. Vol. 13. 2013, pp. 443–458.

[39] Stephen Dawson-Haggerty et al. "sMAP: a simple measurement and actuation profile for physical information". In: *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*. ACM. 2010, pp. 197–210.

[40] DGraph. *BadgerDB*. `https://github.com/dgraph-io/badger`.

[41] Roger Dingledine, Nick Mathewson, and Paul Syverson. *Tor: The second-generation onion router*. Tech. rep. Naval Research Lab Washington DC, 2004.

[42] Colin Dixon et al. "An operating system for the home". In: *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association. 2012, pp. 25–25.

[43] Dylan Dreyer. "A Secure Message Broker in an Untrusted Environment". MA thesis. EECS Department, University of California, Berkeley, May 2019. URL: `http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-91.html`.

[44] Adam Eijdenberg, Ben Laurie, and Al Cutter. *Verifiable Data Structures*. `https://github.com/google/trillian/blob/master/docs/VerifiableDataStructures.pdf`.

[45] Carl M Ellison et al. “SPKI examples”. In: *draft-ietf-spki-cert-examples-01. txt* 10 (1998).

[46] Ksenia Ermoshina, Francesca Musiani, and Harry Halpin. “End-to-End Encrypted Messaging Protocols: An Overview”. In: *INRIA*. 2017.

[47] Victor Manuel Escobedo et al. “BeyondCorp: The User Experience”. In: *Login* (2017).

[48] Ethereum. *Light Ethereum Subprotocol LES*. `https://github.com/ethereum/devp2p/blob/master/caps/les.md`. 2019.

[49] Ethereum. *Solidity*. `https://solidity.readthedocs.io/en/develop/`. 2017.

[50] Evernym Inc. *Everynm: Self-sovereign identity with verifiable claims*. 2018.

[51] Facebook. *RocksDB*. `https://rocksdb.org/`. 2019.

[52] *Facebook permission bug*. `https://web.archive.org/web/20190606001108/https://money.cnn.com/2018/06/07/technology/facebook-public-post-error/index.html`. 2018.

[53] Anna Felkner and Adam Kozakiewicz. “Practical Extensions of Trust Management Credentials”. In: *Advances in Network Systems*. Springer, 2017, pp. 167–180.

[54] Earlence Fernandes et al. “Decentralized Action Integrity for Trigger-Action IoT Platforms”. In: *Proc. NDSS*. 2018.

[55] Philip WL Fong. “Relationship-based access control: protection model and policy language”. In: *Proc. ACM CODASPY*. 2011.

[56] Romain Fontugne et al. “Strip, bind, and search: a method for identifying abnormal energy consumption in buildings”. In: *Proceedings of the 12th international conference on Information processing in sensor networks*. ACM. 2013, pp. 129–140.

[57] The Linux Foundation. *Kubernetes*. 2017. URL: `https://kubernetes.io` (visited on 06/16/2017).

[58] Keith Frikken et al. “Attribute-based access control with hidden policies and hidden credentials”. In: *IEEE Transactions on Computers* 55.10 (2006), pp. 1259–1270.

[59] Sadayuki Furuhashi. *MessagePack encoding*. `https://msgpack.org`. 2019.

[60] Go. *Gob encoding*. `https://golang.org/pkg/encoding/gob/`. 2019.

[61] Dan Goodin. *Already on probation, Symantec issues more illegit HTTPS certificates*. `https://arstechnica.com/security/2017/01/already-on-probation-symantec-issues-more-illegit-https-certificates/`. Jan. 2017.

[62] Dan Goodin. *Google takes Symantec to the woodshed for mis-issuing 30,000 HTTPS certs*. `https://arstechnica.com/security/2017/03/google-takes-symantec-to-the-woodshed-for-mis-issuing-30000-https-certs/`. Mar. 2017.

[63] Google. *Chrome's plan to distrust symantec certificates*. https://web.archive.org/web/20190323041559/https://security.googleblog.com/2017/09/chromes-plan-to-distrust-symantec.html. 2017.

[64] Google. *Key Transparency*. https://github.com/google/keytransparency/blob/master/docs/design.md.

[65] Google. *LevelDB*. https://https://github.com/google/leveldb.

[66] Google. *Protocol buffers*. https://developers.google.com/protocol-buffers/. 2019.

[67] Google. *Trillian*. https://github.com/google/trillian.

[68] Google. *VLBM implementation*. https://github.com/google/trillian/tree/master/examples/ct/ctmapper.

[69] Johannes Götzfried et al. "Cache Attacks on Intel SGX". In: *Proceedings of the 10th European Workshop on Systems Security*. EuroSec'17. Belgrade, Serbia: ACM, 2017, 2:1–2:6. ISBN: 978-1-4503-4935-2. DOI: 10.1145/3065913.3065915. URL: http://doi.acm.org/10.1145/3065913.3065915.

[70] V. Goyal et al. "Attribute-based encryption for fine-grained access control of encrypted data". In: *CCS*. 2006.

[71] OAuth Working Group. *OAuth 2 Token Exchange*. https://tools.ietf.org/html/draft-ietf-oauth-token-exchange-15. 2018.

[72] Rasmus Halvgaard et al. "Economic model predictive control for building climate control in a smart grid". In: *Innovative Smart Grid Technologies (ISGT), 2012 IEEE PES*. IEEE. 2012.

[73] Jason E Holt et al. "Hidden credentials". In: *ACM workshop on privacy in the electronic society*. 2003.

[74] Michael Janosko et al. "BeyondCorp 6: Building a Healthy Fleet". In: *Login* 43 (2018).

[75] Department of Justice. *The Mueller report*. https://www.justice.gov/storage/report.pdf. 2019.

[76] John Kolb. "Spawnpoint: Secure Deployment of Distributed, Managed Containers". PhD thesis. Master's thesis. EECS Department, University of California, Berkeley, 2018.

[77] Krebs On Security. *Target Hackers Broke in Via HVAC Company*. https://web.archive.org/web/20190616025020/https://krebsonsecurity.com/2014/02/target-hackers-broke-in-via-hvac-company/. 2014.

[78] Sam Kumar et al. "JEDI: Many-to-Many End-to-End Encryption and Key Delegation for IoT". In: *CoRR* (2019).

[79] Sam Kumar et al. "Toward a Secure File-Sharing System from Decentralized Trust". In: *In Submission*.

[80] Selena Larson. *Every single Yahoo account was hacked - 3 billion in all*. Ed. by CNN. Online. Oct. 2017. URL: %5Curl%7Bhttp://money.cnn.com/2017/10/03/technology/business/yahoo-breach-3-billion-accounts/index.html%7D.

[81] Ben Laurie. *Revocation Transparency*. https://www.links.org/files/RevocationTranspa pdf. 2018.

[82] Ben Laurie, A Langley, and E Kasper. *Certificate transparency (RFC 6992)*. 2013.

[83] David Lazar. *Open-source IBE implementation, part of the Vuvuzela project*. https://github.com/vuvuzela/crypto.

[84] Ninghui Li et al. "Design of a role-based trust-management framework". In: *IEEE S & P*. 2002.

[85] Ninghui Li et al. "Distributed credential chain discovery in trust management". In: *Journal of Computer Security* 11.1 (2003), pp. 35–86.

[86] Ninghui Li and John C Mitchell. "Datalog with constraints: A foundation for trust management languages". In: *International Symposium on Practical Aspects of Declarative Languages*. Springer. 2003, pp. 58–73.

[87] Benoît Libert and Jean-Jacques Quisquater. "Identity based encryption without redundancy". In: *ACNS*. Springer. 2005.

[88] Marcela S Melara et al. "CONIKS: Bringing Key Transparency to End Users." In: *USENIX Security Symposium*. Vol. 2015. 2015, pp. 383–398.

[89] Sascha Müller and Stefan Katzenbeisser. "Hiding the policy in cryptographic access control". In: *International Workshop on Security and Trust Management*. Springer. 2011, pp. 90–105.

[90] Deirdre K. Mulligan and Helen Nissenbaum. *The Concept of Handoff as a Model for Ethical Analysis and Design*.

[91] *OAuth 2.0*. https://oauth.net/2/. 2018.

[92] Barclay Osborn et al. "BeyondCorp: Design to Deployment at Google". In: *Login* 41 (2016), pp. 28–34. URL: https://www.usenix.org/publications/login/spring2016/osborn.

[93] Christopher Palmer et al. "Mortar. io: Open Source Building Automation System". In: *BuildSys-ACM Int. Conf. on Embedded Systems for Energy-Efficient Built Environments*. 2014, pp. 204–205.

[94] Joseph Poon and Thaddeus Dryja. *The bitcoin lightning network: Scalable off-chain instant payments*. 2016.

[95] Ronald L Rivest and Butler Lampson. "SDSI-a simple distributed security infrastructure". In: Crypto. 1996.

[96] Anthony Rowe et al. “Sensor Andrew: Large-scale campus-wide sensing and actuation”. In: *IBM Journal of Research and Development* 55.1.2 (2011), pp. 6–1.

[97] Scalagent. *Benchmark of MQTT servers*. `http://www.scalagent.com/IMG/pdf/Benchmark_MQTT_servers-v1-1.pdf`. 2015.

[98] Martin Schanzenbach et al. “Practical Decentralized Attribute-Based Delegation using Secure Name Systems”. In: *arXiv:1805.06398* (2018).

[99] Kent E Seamons et al. “Requirements for policy languages for trust negotiation”. In: *International Workshop on Policies for Distributed Systems and Networks*. IEEE. 2002, pp. 68–79.

[100] Adi Shamir. “How to share a secret”. In: *Comm. ACM* (1979).

[101] Chenguang Shen et al. “Beam: ending monolithic applications for connected devices”. In: *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association. 2016, pp. 143–157.

[102] Yonatan Sompolinsky and Aviv Zohar. “Accelerating Bitcoin’s Transaction Processing. Fast Money Grows on Trees, Not Chains.” In: *IACR Cryptology ePrint Archive* 2013.881 (2013).

[103] Batz Spear et al. “Beyond Corp: The Access Proxy”. In: *Login* (2016).

[104] Mudhakar Srivatsa and Mike Hicks. “Deanonymizing mobility traces: Using social network as a side-channel”. In: *ACM conference on Computer and communications security*. 2012, pp. 628–637.

[105] Peter Szilagyi. *eth/Fast Synchronization Algorithm*. `https://github.com/ethereum/go-ethereum/pull/1889`. 2015.

[106] Jay Taneja et al. “Enabling advanced environmental conditioning with a building application stack”. In: *Green Computing Conference (IGCC), 2013 International*. IEEE. 2013, pp. 1–10.

[107] The Sovrin Foundation. *A Protocol and Token for Self-Sovereign Identity and Decentralized Trust*. 2018.

[108] Vamsi Thummala and Jeff Chase. “SAFE: A declarative trust management system with linked credentials”. In: *arXiv preprint arXiv:1510.04629* (2015).

[109] New York Times. *Facebook and Cambridge Analytica: What You Need to Know as Fallout Widens*. `https://www.nytimes.com/2018/03/19/technology/facebook-cambridge-analytica-explained.html`. 2018.

[110] Daniel Trivellato et al. “GEM: A distributed goal evaluation algorithm for trust management”. In: *Theory and practice of logic programming* 14.3 (2014), pp. 293–337.

[111] Rory Ward and Betsy Beyer. “BeyondCorp: A New Approach to Enterprise Security”. In: *;login:* Vol. 39, No. 6 (2014), pp. 6–11.

[112] Nico Weichbrodt et al. “AsyncShock: Exploiting synchronisation bugs in Intel SGX enclaves”. In: *European Symposium on Research in Computer Security*. Springer. 2016, pp. 440–457.

[113] Marianne Winslett et al. “Negotiating trust in the Web”. In: *IEEE Internet Computing* 6.6 (2002), pp. 30–37.

[114] WIRED. *The 21 (and Counting) Biggest Facebook Scandals of 2018*. `https://www.wired.com/story/facebook-scandals-2018/`. 2018.

[115] XMPP Standards Foundation. *XMPP*. `https://xmpp.org`. 2017.

[116] Mahdi Zamani, Mahnush Movahedi, and Mariana Raykova. “RapidChain: Scaling blockchain via full sharding”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM. 2018, pp. 931–948.

[117] Xian Zhu et al. “Distributed credential chain discovery in trust-management with parameterized roles”. In: *International Conference on Cryptology and Network Security*. Springer. 2005, pp. 334–348.