WAVE: A Decentralized Authorization Framework with Transitive Delegation

Michael P Andersen, Sam Kumar, Moustafa AbdelBaky
Gabe Fierro, John Kolb, Hyung-Sin Kim, David E. Culler, Raluca Ada Popa
University of California, Berkeley

Abstract

Most deployed authorization systems rely on a central trusted service whose compromise can lead to the breach of millions of user accounts and permissions. We present WAVE, an authorization framework offering *decentralized trust*: no central services can modify or see permissions and any participant can delegate a portion of their permissions autonomously. To achieve this goal, WAVE adopts an expressive authorization model, enforces it cryptographically, protects permissions via a novel encryption protocol while enabling discovery of permissions, and stores them in an untrusted scalable storage solution. WAVE provides competitive performance to traditional authorization systems relying on central trust. It is an open-source artifact and has been used for two years for controlling 800 IoT devices.

1 Introduction

Authorization and authentication are fundamental components of many systems. Most authorization systems today *rely on centralized services* such as centralized credential stores (e.g., [15, 19, 56]), Access Control Lists (ACLs), Active Directory, and OAuth [4]. For example, in a calendar application, a central service stores which users have access to what calendars, and users authenticate to it, e.g. via username and password. In these systems, delegation is critical: for instance, allowing an assistant to edit your calendar, and letting the assistant further delegate restrictive view access to an event organizer. These forms of delegation are typically implemented as changes to a centralized ACL.

However, this approach presents two fundamental problems. First, a centralized service is a central point of attack: a single attack can simultaneously compromise many user accounts and permissions. There have been numerous such breaches [39], and attackers even managed to log in as the victim users. Second, the operator of the central server has a complete view of the private permission data for all users (thus seeing users' social relationships [54]), and can modify permissions [2].

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 encryption systems [25], such as WhatsApp and Signal, blockchains (e.g., Bitcoin, Ethereum, Zcash), or ledgers (e.g., IBM's Hyperledger [17], Certificate Transparency [41], Key Transparency [32]). Our goal is to build a scalable decentralized authorization system, permitting delegation under a similar threat model.

We propose a decentralized authorization system that *does not rely on a trusted service*, *WAVE ("**W**AVE is an **A**uthorization **V**erification **E**ngine")*. WAVE offers decentralized trust: each user's WAVE client manages the permissions of that user and can delegate access to other users. WAVE enforces delegation *cryptographically*, not via a trusted service. It aims to capture a wide range of authorization policies and to provide an alternative to traditional systems, such as OAuth [4] and Active Directory.

Importantly, in providing decentralized transitive delegation, WAVE facilitates applications that span multiple trust domains. For example, IoT orchestration applications like If This Then That (IFTTT) [3] tie together multiple vendors and users, but IFTTT's design relies on several central points of attack: the vendor OAuth servers and the IFTTT token storage servers. The compromise of any one of these servers may affect hundreds of thousands of users. Using WAVE, greater cross-administrative-domain orchestration can be achieved with no central authorization servers, reducing the trust that each domain must place in the others.

1.1 Usage Scenarios

While authorization plays a key role in the security of almost any system today, the benefits of decentralized authorization are most pronounced in systems that are inherently distributed, where the prevailing centralized authorization schemes undermine what would otherwise be a resilient system. Our deployment of WAVE over the past two years has focused on securing distributed IoT devices and services used to monitor and control over twenty small to medium-sized commercial and residential buildings; hence, we will use smart buildings as a running example.

Consider a set of campuses, each owned by a property manager. Each campus is composed of multiple buildings, with portions of each building leased out to tenants by the property manager. The property manager within each campus is the authority for the cyberphysical resources associated with the buildings in the campus, but they must *delegate* permission to the individual building managers who must further delegate permissions to the tenants, allowing them to control the portions of the buildings that they rent. Any of these principals may then further delegate permissions to IoT devices, long-running analytics or control services operating on their behalf, perhaps provided by the utility. The building manager and/or tenant will also grant ephemeral permissions on subsets of the building infrastructure to contractors (like HVAC commissioning teams) and, especially in our case, to

researchers.

A similar structure occurs in small residential buildings where a homeowner installs smart devices such as lights and thermostats and needs to delegate permission on those devices to their partner, guest, nanny, or children.

Cross-administrative-domain delegation is present in both examples. In larger buildings, we see the boundary between the property owner and the tenants. In residential buildings, this is most evident when using orchestration tools like IFTTT, where an organization, distinct from the owner of the devices, runs the controller service and needs to obtain permission from the owner.

WAVE is not limited to IoT. It provides general purpose delegable authorization and can, for example, be used in place of OAuth to remove the risk of the centralized token-issuing server and allow for richer delegation semantics.

1.2 High-Level Security Goal & Threat Model

At a high level, our objective is to design a system where the compromise of an authorization server does not compromise all the users' permissions. 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 §4 and §B 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.

1.3 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 IoT usage scenarios. More concretely, we summarize the following six requirements that are not simultaneously met by any existing system (as illustrated in Table 4):

No reliance on central trust. 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. In this case, 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. In contrast, this property is well developed in academic work [49, 13, 43, 45, 29, 14, 51, 11].

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). 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. We elaborate further in §9.

Decentralized verification. Some existing decentralized systems (e.g. SDSI/SPKI [49] and Macaroons [12]) 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 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 [12]), 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 [49], Macaroons [12] and [13, 43, 45, 29, 27, 44, 59, 18, 57, 50]).

While many existing systems meet some of these require-

ments, no existing work meets all of the requirements concurrently, as shown in §9.

1.4 Challenges and Approach

Compatible authorization model. The first challenge is identifying a model for authorization that is compatible with these requirements. We examined many authorization models [12, 49, 24, 13, 43, 45, 29, 27, 58, 37, 30, 48, 19, 15, 56, 44, 59, 18, 57, 50], but most of them cannot be enforced without a centralized authority or are incompatible with the other requirements. Nevertheless, we found that representing the authorization model as a graph, such as in SDSI/SPKI [49, 24] where a proof of authorization is a path through a graph, is compatible with our requirements, even though the existing systems implementing it fall short.

Consequently, WAVE maintains a global graph of delegations between entities (Fig. 1a), which are associated with participants. An *entity* is a collection of public and private key pairs and can correspond to a user, service, or group. An edge indicates that an entity grants another entity access according to a *policy*, which is one or more permissions along with a description of the resources for which the permissions are granted, and the expiry of the grant. This enables fine-grained transitive delegation with revocation and expiry.

To enforce the policy cryptographically, each edge, from *issuer* to *subject* entity, is a signed certificate recording the delegation of permissions, which we call an *attestation*. A path from an entity to another entity grants access equal to the *intersection* of the policies on that path. The graph enables entities to *prove* they have some permission `P` by revealing a path through the graph from an authority entity to themselves where all the edges of the path grant a superset of `P`. This path is called a *proof*. The graph construction allows permissions to be granted in any order, including delegation of permissions one does not yet possess but expects to receive in the future.

While WAVE's authorization graph and proofs are structurally similar to SDSI/SPKI, WAVE differs in three important aspects: (1) while in SDSI/SPKI only a central authority (holding an ACL) can verify a proof, in WAVE anyone can independently (with no communication) verify a proof yielding an authorization policy. (2) WAVE provides a trustworthy, scalable storage solution for attestations that enables discoverability with offline participants and out of order grants, which is out of scope for SDSI/SPKI. (3) Attestations are encrypted in WAVE whereas they are visible in SDSI/SPKI. These differences enable meeting the requirements in §1.3.

Scalable untrusted storage. To support granting permissions to offline participants, we use a storage system that enables participants to discover attestations when they later come online. To meet the requirements above, the storage must be able to prove its integrity cryptographically, so as not to compromise Security Goal #3.

Our first design of WAVE [9] was built on Ethereum, which has these properties. Unfortunately, our experiments showed that a blockchain-based system will not scale to a global size, even though changing permissions is far less common than accessing data.

We present a new type of transparency log, the *Unequivocable Log Derived Map (ULDM)*. Unlike Certificate Transparency [41], which cannot form a proof of nonexistance needed for revocations, or Key Transparency [32], which requires users to audit every object at every epoch, a ULDM is both capable of handling revocations and is efficiently auditable. The ULDM forms the foundation of a horizontally scalable storage tier with cryptographically proven integrity, which could also be useful outside of WAVE. Our current design, described in §5, allows for a shared-nothing architecture of storage servers with independent auditors that need only communicate periodically (e.g., once a day) with clients to verify the correct operation of the storage. The resulting architecture is arbitrarily horizontally scalable with each node having a higher capacity and lower latency than a blockchain, as we show in §8.

Confidentiality of permissions. To meet the requirement of protected permissions and Security Goal #2 despite the public ULDM storage tier, there must be a mechanism to prevent the storage servers or the general public from seeing the permissions, while ensuring that parties forming and verifying proofs can see the necessary permissions. The challenge lies in preserving confidentiality while enabling out of order delegation and offline participants. We overcome this challenge with a novel technique called *reverse-discoverable encryption* (RDE, §4) used to encrypt attestations. RDE allows entities to efficiently discover and decrypt the attestations that they can use in a valid proof while using policy-aware encryption to hide most other attestations. Importantly, RDE does not introduce additional constraints on the ordering of delegations or liveness of participants.

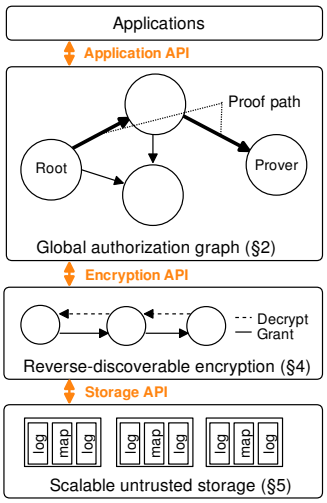
Our implementation of WAVE is a real-world *open-source* artifact [7]. We have deployed and operated various versions of WAVE over the past two years. During this time, WAVE has been used to control more than 20 buildings containing more than 800 IoT devices. We discuss lessons from our deployment in §8.4; in particular, this has allowed us to confirm that the authorization and delegation model presented here is useful in practice. Further, WAVE has offered performance comparable to traditional authorization systems, validating real proofs in 1–4 ms, depending on the depth of delegation.

2 WAVE Overview

WAVE runs as a service that can be logically divided into three layers (Fig. 1a) each providing an API (Fig. 1b).

2.1 Global Authorization Graph

Recall that the global authorization graph consists of entities, which are bundles of public and private keys, and attestations, which are the permission grants between them. The client (representing a user, device, or service) inter-



(a) The WAVE stack

Subsystem	API
Application	CreateEntity() ⟹ (privEnt, pubEnt)
	Delegate(issuer:privEnt, subject:pubEnt, policy) ⟹ attestation
	CreateProof(subject:privEnt, policy) ⟹ (proof)
	VerifyProof(proof) ⟹ (subject:pubEnt, policy)
	NewName(issuer: privEnt, subject:pubEnt, name) ⟹ (nameDecl)
	ResolveName(resolver: privEnt, name) ⟹ (nameDecl)
	Revoke(issuer:privEnt, object:attestation/pubEnt)
Encryption	EncryptAttest(attestation, partition) ⟹ attCiphertext
	DecryptAttest(perspective:privEnt, attCiphertext) ⟹ attestation
Storage	Put(object, server) ⟹ hash
	Get(hash, server) ⟹ object
	Enqueue(list:hash, entry:hash, server)
	IterQueue(list:hash, cursor) ⟹ (entry:hash, newCursor)

(b) The API provided by WAVE's stack.

Figure 1: An overview of WAVE

acts through the WAVE service with the global authorization graph. Clients can create new entities (e.g., for a service they are deploying).

To grant permissions to other entities, clients use the WAVE service to construct an attestation signed by the granting entity containing a policy describing the permissions. An attestation A consists of:

- A.issuer: the entity that wishes to grant permissions to another entity,
- A.subject: the entity receiving the permissions,
- A.policy: an expression of permissions, for example, RTree described in §2.4, and
- a revocation commitment described in §6.1
- signature(s) from the issuer.

When accessing a service or controlling a device, clients request a proof from the WAVE service; the WAVE service will search for a path through the global authorization graph from the authority for the service or device in question to the client's entity, where each edge grants a superset of the required permissions. The representation of this path is a self-standing proof of authorization that can be verified without communication with the proving entity. The receiving service or device can use the WAVE service to validate a proof, yielding the authorization policy it permits.

The WAVE service also allows for mapping human readable names to entity public keys to make the system more usable, as we elaborate in §6.2.

2.2 Reverse-Discoverable Encryption (RDE)

To ensure the privacy of permissions, the WAVE service uses our protocol, Reverse-Discoverable Encryption (described in §4) to encrypt the attestations. The encryption layer is transparent to clients: the WAVE service will discover and decrypt the portion of the global graph that concerns the client automatically. The only time a client interacts with the encryption layer is when they use RDE to encrypt messages for application-level end-to-end encryption, which is beyond the scope of this paper.

2.3 Scalable Untrusted Storage

When the client instructs the WAVE service to create an entity or an attestation, the WAVE service will place the public keys (for entities) or RDE ciphertext (for attestations) into the scalable untrusted storage (§5). As with RDE, the placement into storage is transparent to clients: clients operate only at the level of granting permissions, creating proofs and verifying proofs. The WAVE client will interact with the storage to discover and decrypt the portion of the global graph necessary for performing those actions without the client manually publishing or retrieving objects.

2.4 Resource Tree Authorization Policy

Although WAVE's design is agnostic to the specific mechanism used for expressing the authorization policy (i.e., it is compatible with existing policy languages such as [10, 12]), in our IoT deployments we use a simple yet widely applicable model: a resource tree (*RTree*) modelled roughly after SPKI's pkpfs tags [24].

An RTree policy manages permissions on a hierarchically organized set of resources. A resource is denoted by a URI pattern such as `company-entity/building/device` or `user-entity/albums/holiday/*`. The first element of a URI (e.g. `company-entity`) is called the *namespace authority* or just *namespace*, which specifies the entity who is the *root of authorization* for that resource (the entity who has permission on that policy without having received permission from someone else). The global authorization graph has many different RTrees with namespace authorities, ideally with one per intrinsic authority, e.g. homeowner or company. This lets the system be as decentralized as the naturally occurring authority structure, unlike the single-authority-per-vendor model, used in most systems today, which forces centralization. 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*. 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

For example, a building manager entity might grant `hvac::actuate` on `bldgnamespace/floor4/*` over a time range corresponding with the lease terms, allowing further delegation, to a tenant entity.

2.5 How WAVE Meets the Requirements

WAVE's global authorization graph, RDE, and storage layer allow it to achieve the requirements established in §1.3:

No reliance on central trust. WAVE achieves decentralization via three design features. First, the permission delegations are cryptographically enforced without a verifying authority. Secondly, any participant can create an RTree namespace, mimicking the natural ownership of resources. Finally, our Unequivocable Log Derived Map §5 allows participants to detect if the untrusted storage servers violate integrity. Although the storage server is centralized for availability, it is not a point of central trust as its behavior is cryptographically enforced.

Transitive Delegation. The graph-based authorization model efficiently captures transitive delegation. To delegate permissions, any entity can create an attestation that captures which subset of their permissions they wish to delegate. Since a proof is represented by a path through the graph, if an entity higher up in the delegation tree is revoked, all entities beneath it will no longer be able to prove they have permissions, even though the party revoking the entity may have been unaware of the delegations lower in the tree. This gives us the transitive delegation property.

Protected permissions. Through the Reverse-Discoverable Encryption scheme in §4, no party can decrypt attestations that are not potentially relevant to them. In our example, the property manager cannot decrypt attestations that the tenant makes, and the party running the storage servers cannot read any of the attestations.

Decentralized verification. WAVE proofs can be verified by anyone, unlike in SDSI/SPKI [49] or Macaroons [12]. This enables an IoT device to verify all messages it receives without communicating with an external service (with the exception of revocation checks, as detailed in §6.1).

No ordering constraints. An entity can grant any permissions at any time, including those that it has not yet received (although the recipient won't be able to form a proof yet). Consequently, attestations can be replaced anywhere in the hierarchy without requiring re-issue of subsequent delegations. Furthermore, our privacy mechanism preserves this property because an attestation can be encrypted under a policy-specific key before the issuer has been granted the permissions corresponding to the policy.

Offline participants. Attestations are disseminated through the ULDM storage tier (§5) which allows for entities to discover permissions they have been granted while they were offline and removes the need for any out-of-band online communication between entities.

3 Security Guarantees and Roadmap

WAVE must fulfill three security goals (§1.2). Regarding Security Goal #1, WAVE guarantees the following:

Guarantee 1. *An attacker* Adv *can form a proof of authorization on a policy if and only if the authority for that policy is compromised or has delegated access, directly or indirectly, to a compromised entity.*

This guarantee follows directly from the fact that each attestation is signed by its issuer. A WAVE proof can be thought of as a certificate chain. Given that existing systems like SDSI/SPKI [49] use a similar construction, we do not explore this further.

To achieve the other two security goals, WAVE introduces two new techniques: Reverse-Discoverable Encryption (§4) to satisfy Security Goal #2, and Unequivocable Log-Derived Maps (§5) to satisfy Security Goal #3. The following sections introduce these techniques and state formal security guarantees.

4 Encrypting Attestations

We encrypt attestations such that entities can decrypt attestations they can use in a valid proof. Entities cannot learn the policy (i.e., what permissions are granted) or the issuer (i.e., who created the attestation) of most other attestations. Our technique, *reverse-discoverable encryption* (RDE), does not require out-of-band communication between entities and works even if attestations are created out of order.

We present our solution incrementally: §4.1 formalizes the problem that RDE solves. §4.2 presents a simplified design of RDE, based on traditional public-key encryption, that provides a weak but useful security guarantee called "structural security." §4.3 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. §4.4 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, we assume that the attacker Adv is computationally-bounded, and that standard cryptographic assumptions hold.

4.1 Graph-based Formalization

We formalize the problem in terms of the global authorization graph; an example is shown in Fig. 2. For **correctness**, we require that each entity can decrypt all attestations that

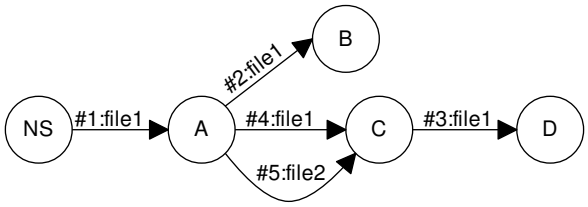


Figure 2: The number to the left of each colon indicates when the attestation was created. The string to the right denotes the resource on which it grants permission.

it can use to form a valid proof where it is the subject. In Fig. 2, 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.

4.2 Structural RDE

This section explains a simplified (yet 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 its 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. 2, #3 contains sk_C and is encrypted under pk_D (i.e., $\mathrm{Enc}(\mathrm{pk}_D; \#3||\mathrm{sk}_C)$).

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 A for which there exists a path from $A.\mathsf{subject}$ to e. We call it "structural" security because only the structure of the graph, not the policies in attestations, affects whether A is visible to e. While structural RDE uses traditional public-key encryption, it differs from systems like PGP in that entities include their long-lived private keys in the attestations they encrypt.

4.3 Policy-Aware RDE

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). 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*. In particular, traditional public-key encryption is insufficient for policy-aware encryption (except for a boolean policy). We use the RTree policy type to explain our policy-aware RDE, although the technique applies to other policy types.

We identify Wildcard Identity-Based Encryption (WIBE) [5] as a suitable policy-aware encryption scheme to implement RDE for the RTree policy type. Typically, IBE [16] (or an IBE variant such as WIBE) 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 WIBE system for every entity, so there is no central PKG*.

A WIBE system consists of a master secret and public key pair $(\mathsf{WIBE.msk}, \mathsf{WIBE.mpk})$. A message m is encrypted using the master public key $\mathsf{WIBE.mpk}$ and a fixed-length vector of strings, called an ID: $\mathsf{WIBE.Enc}(\mathsf{WIBE.mpk}, \mathsf{ID}; m)$. 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, $\mathsf{WIBE.Enc}(\mathsf{WIBE.mpk}, \mathsf{ID}; m)$, if ID^* and ID match in all non-wildcard components.

Every policy p has an associated WIBE ID called a *partition*. The partition corresponding to policy p is denoted $P(p)$. When issuing an attestation A, an entity encrypts it using $P(A.\mathsf{policy})$, in the WIBE system of $A.\mathsf{subject}$: $\mathsf{WIBE.Enc}(\mathsf{WIBE.mpk}_{A.\mathsf{subject}}, P(A.\mathsf{policy}); A)$. Furthermore, the issuing entity generates secret keys in its own WIBE 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 $P(p)$ for p compatible with $A.\mathsf{policy}$, then A includes $W = \{\mathsf{WIBE.KeyGen}(\mathsf{WIBE.msk}_{\mathsf{Issuer}}; \mathsf{ID}^*{}_i)\}_{\mathsf{ID}^*{}_i \in Q(A.\mathsf{policy})}$. Below, we develop the *partition map* for RTree, which derives a partition from an RTree policy (i.e., functions P and Q).

Partition map for RTree. To define P, consider that an RTree policy consists of a resource prefix as defined in §2.4 (matching multiple resources) and a time range during which

the permission is valid. To express the start and end of this range as a WIBE 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 WIBE 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. When granting an attestation for an RTree policy, the issuer encrypts the attestation contents under the resulting WIBE $\mathsf{ID} = P(A.\mathsf{policy})$. Note that for a time tree of depth k, and a resource prefix of length ℓ, WIBE must be instantiated with at least $n = 2k + \ell$.

The issuer must also include the policy-specific WIBE keys from their own system in the attestations, generated with ID^*s $Q(A.\mathsf{policy})$, 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 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.

4.4 Efficient Discoverability

In the scheme above, attestations are encrypted under the partition in the subject's WIBE system. Unfortunately, it is subject to two major shortcomings. First, a WIBE 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. 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 [46, 42] because we do not need wildcards. As with the WIBE 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 WIBE ciphertext under this IBE encryption so that the partition cannot be extracted. Finally, we include IBE keys from the issuer's IBE system, to allow the subject to discover the partition of upstream attestations. We denote the ID^*s corresponding to these keys as $M(A.\mathsf{policy})$. Because the partition label is simpler in structure than the partition, defining $M(A.\mathsf{policy}) = \{L(A.\mathsf{policy})\}$ is sufficient. So far, what gets stored in the attestation is:

$$\begin{aligned}&\mathrm{IBE.Enc}(\mathrm{IBE.mpk}_{A.\mathsf{subject}}, L(A.\mathsf{policy}); P(A.\mathsf{policy}) \,||\\ &\quad \mathrm{WIBE.Enc}(\mathrm{WIBE.mpk}_{A.\mathsf{subject}}, P(A.\mathsf{policy}); W \,||\, I))\end{aligned} \tag{1}$$

where W is defined as above, and

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

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

4.5 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 B. 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 the identifier of the key 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 B.

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 [1], a proxy, or other anonymous/secure messaging methods [21]).

Revocation. Although revoked attestations cannot be used in a proof due to the commitment revocation scheme described in §6.1, they still confer the ability to decrypt upstream attestations. Therefore we consider them part of the graph in the formal guarantees (Appendix B). 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. 2 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.

4.6 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 WIBE 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 *compartments*, one for the prover (that includes keys it needs to decrypt other attestations) and one for both the prover and the verifier (that includes the policy, 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 with WIBE. This allows the prover to reveal to the verifier the necessary parts of an attestation by sending it the AES verifier key, without allowing the verifier to decrypt other attestations. The final structure of the attestation is in Fig. 3.

4.7 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

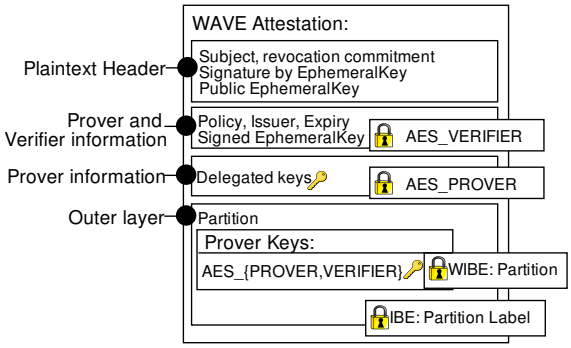


Figure 3: Encrypted WAVE attestation structure. The locks indicate the key used to encrypt the content.

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 WIBE 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 WIBE 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 WIBE 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.

4.8 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 B). This can be marginally improved by including an additional set of WIBE keys in the attestations to allow for the full resource (not just the prefix) to be captured by P and Q but this increases the number

of included keys by a factor of ℓ. Additionally, using KP-ABE [35] instead of WIBE would result in smaller attestations, but higher decryption times.

Aside from different encryption schemes, the RDE technique also generalizes beyond the RTree policy described above. Careful selection of P and Q, coupled with the use of a more expressive encryption scheme such as KP-ABE [35] allows for the realization of a more expressive policy (e.g. those discussed in §9) at the cost of decreased performance. While we have not found this trade-off warranted in our setting, this extension is straightforward and still meets our security goals. The formalism in Appendix B largely generalizes to other policy types, but the semantics of compatibility (Note 1) will change depending on the encryption schemes used and on the choice of P, Q, L, and M.

5 Scalable Untrusted Storage

To avoid centralized trust when storing attestations, we contribute a storage tier that enforces integrity cryptographically. This tier is physically decentralized: it is spread over multiple servers owned by different parties. Importantly, these individual servers are trusted to maintain availability, but not integrity (in the spirit of Certificate Transparency [41]) or privacy (achieved by RDE, §4). Thus, users and services can interact with storage servers that anybody operates, without trusting the servers' operators, except for availability.

The storage API (Fig. 1b) consists of four functions: Get and Put are used for placing/retrieving entities, attestations, name declarations (§6.2) and revocation secrets (§6.1) in storage; Enqueue places an object hash at the end of a named queue, and IterQueue allows retrieval from a queue. The queue functions 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.

A blockchain is a natural candidate for such a storage tier. Multiple servers are responsible for maintaining a blockchain, and, due to the underlying Merkle tree data structure, any one server can prove the integrity of its responses to state queries according to a specific Merkle tree root hash, meeting the requirements.

Prior versions of WAVE used an Ethereum blockchain, but extended use and experimentation revealed this solution to be inadequate for three reasons: (1) A blockchain introduces significant latency when adding objects to storage (up to a minute for a confirmed addition in Ethereum). (2) Participating in a blockchain requires constant network bandwidth and CPU time. (3) The blockchain does not scale past a few tens of transactions per second [22], so it could not store attestations for a global authorization system permitting thousands of delegations per second.

Although this problem appears solvable with existing transparency logs such as Certificate Transparency (CT) [41] or Key Transparency (KT) [32], neither of those is appropriate. CT cannot efficiently prove an object does not exist, needed for revocations, and KT is not efficiently auditable in our context (§9).

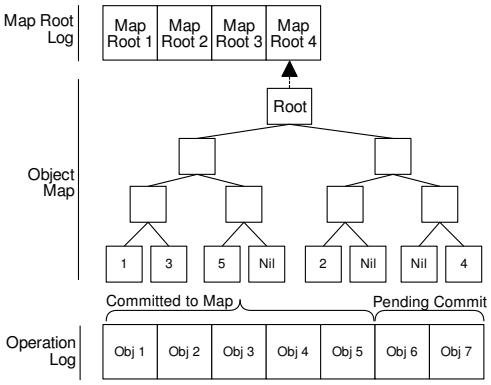


Figure 4: An Unequivocable Log Derived Map (ULDM) built from two Merkle tree logs and a Merkle tree map

Instead, we propose an *Unequivocable Log Derived Map (ULDM)*, a transparency log based on the Verifiable Log Backed Map (VLBM) [23]. 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 [34] 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. 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 [23] as it only stores the hashes of the 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 §5.4.

In what follows, for every reply that the storage server provides, the storage server provides a signature on the reply along with the relevant version of the Map Root Log.

5.1 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 that includes the hashes of the new entries. (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.

5.2 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.

5.3 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 doesn't 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.

5.4 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 [34]), 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.

5.5 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 2 (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 in-*

consistency 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.*

We provide a proof sketch in Appendix A.

6 Revocation and Naming

With the functionality of RDE and ULDM's, we can easily construct a revocation scheme and a PKI-replacing entity naming scheme.

6.1 Commitment-Based Revocation

When a user creates an attestation, it derives a random revocation secret s from a seed stored with the entity private keys and includes a cryptographic hash of s, $\mathsf{hash}(s)$, called the *revocation commitment*, in the attestation. The user then inserts the attestation into ULDM storage. Later on, the user can revoke the attestation by publishing the revocation secret s to the same storage. Revocation of entities works similarly. An entity must have their private key to perform revocation; mechanisms such as [53] 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. WAVE ensures revocation only after the Merge promise deadline. The security of this procedure relies on the guarantees of our ULDM transparency log (§5). Alternatively, the entity forming the proof can include proofs of nonexistence, signed by the storage tier with a timestamp, with the attestations, so that the verifier does not have to perform this lookup.

6.2 Secure Lookup of Public Keys

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 [49]. 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. By traversing this graph, an entity can resolve hierarchical names. For example, consider when an entity representing a company `ACME` names an entity representing a department `Marketing`, which in turn names an entity held by an employee `Alice`. Then, by verifying the identity of a single entity out of band (the company), an entity can resolve the names of all employees within the company's departments, such as `Alice.Marketing.ACME`, without having to manually establish the validity of individual employee entities.

The functionality above, proposed by SDSI, does 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 (§5) to ensure name declarations are discoverable without compromising on the goals of the system (especially without requiring on-line participants). Secondly, WAVE uses a variation of the encryption scheme described in §4 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.

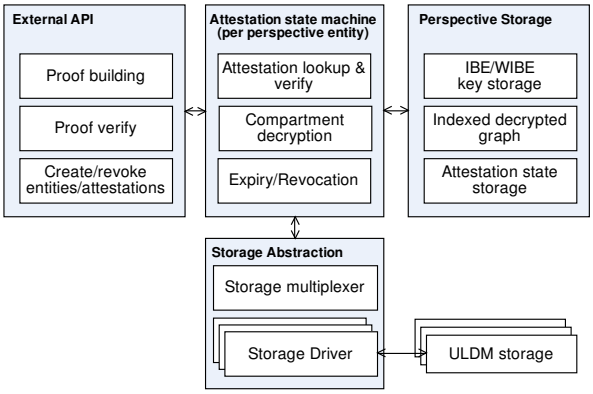


Figure 5: Overview of WAVE's implementation.

7 Implementation

WAVE is implemented in Go and released as open source [7]. It runs as a background service and applications connect via IPC. The service is composed of four logical parts (Fig. 5).

The storage abstraction permits multiple distinct storage providers operating in parallel. As long as the provider implements the API discussed in §5, WAVE can use it. Each storage *driver* is responsible for ensuring the storage is trustworthy, e.g. for a ULDM-based storage it must verify the proofs given by the remote storage server. Attestations can span storage media, i.e., an entity residing on one server can grant permissions to an entity on a different server. We implemented the ULDMs using Merkle trees in Trillian [33] backed by MySQL.

The perspective storage keeps track of the decrypted attestations that form the *perspective graph*. This is the portion of the global graph visible from the perspective of the proving entity. WAVE indexes it to allow efficient key retrieval

Operation	AMD64	ARMv8
Create attestation[1]	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 1: Object operation times [ms].

based on a new attestation and efficient attestation retrieval based on a new key. The index also allows for efficient proof building: finding attestations granted from a given issuer that match specific permissions.

The state machine is responsible for transitioning the attestation through the states of decryption following the discovery process described in §4.7.

The external API is a GRPC [31] API that listens for connections from applications and allows them to use the application API functions given in Fig. 1b. GRPC can generate bindings for multiple languages, so we expect that applications can be written in any language.

The proof builder, when asked to build a proof, begins at the namespace authority (the entity that created the RTree namespace) for the resource that permissions are being proved on, and then performs a shortest path discovery through the perspective graph terminating at the proving entity. Note that this is the opposite direction that attestations are traversed during discovery. Only edges granting a superset of the required permissions are traversed and the maximum depth of traversal is limited by the indirections parameter in the traversed attestations. These two filters make proof building fast for common cases (see §8.1).

8 Evaluation

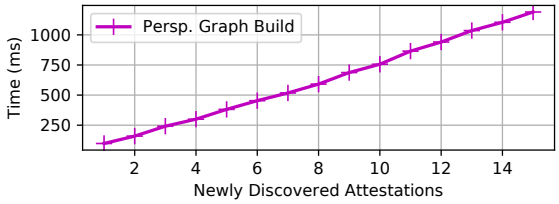
Despite relying on cryptography for its security guarantees, WAVE remains performant, competitive to traditional authentication and authorization systems.

8.1 Microbenchmarks

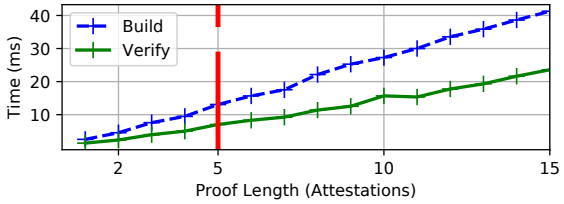
WAVE's performance is dominated by the cost of the core cryptographic operations, shown in Table 1. These are the times measured by a client using the GRPC application API. The measurement is on an Intel i7-8650U AMD64 CPU representative of a standard modern laptop, and on a Raspberry Pi 3, indicative of a low-cost IoT-class ARMv8 platform.

The verifier does not perform any WIBE decryption, as it has the AES verifier key. The subject entity (the direct recipient of the attestation) can skip the IBE decryption of the partition, but must still perform WIBE decryption. Any other entity that is interested in the attestation because it lies further up the delegation chain must perform IBE decryption, WIBE decryption, and then AES decryption. These decryption operations take place only once—when an attestation is added to the perspective graph—so are a one-off cost of receiving permissions. The verifier decryption happens once per unique proof; after that, it is cached so that subsequent verifications complete in negligible time.

[1]Create attestation uses multiple cores



(a) Perspective graph update/build time



(b) Proof build/verification time

Figure 6: Single core timings for proof operations. Vertical line in Fig. 6b is the expected maximum proof length for common applications.

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

Table 2: Latency of LDAP+MySQL, OAuth2 vs. WAVE.

When decrypting attestations and building the perspective graph, we also need to index all the obtained keys and store them on disk. We can see the cost of decryption combined with indexing by measuring the time taken to update a perspective graph, for different sizes of changes to the graph, as shown in Fig. 6a. This includes the time taken to retrieve the encrypted ciphertexts from ULDM-based storage. The dashed vertical line is likely the maximum number of attestations that will be found in a proof as more than five delegations, although supported, is rare in all our deployments.

8.2 Traditional Authorization Flow

To compare WAVE against a traditional authorization system, we benchmark the time taken by a representative backend to turn a username and password into an authorization 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). We also add the time taken to verify an OAuth2 JWT token containing the authorization policy in the form of scopes.

The results are shown in Table 2. 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 con-

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

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

sists of three attestations, the WAVE verification is about half the time of the single-delegation LDAP flow.

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 [36], 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.

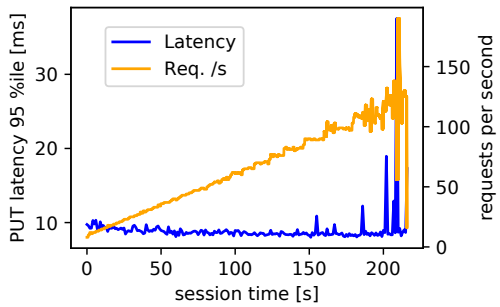
8.3 Storage Evaluation

Since an entity in WAVE does not communicate with any other entity, except via the storage, WAVE's scalability depends on the performance of the global storage. As mentioned in §5, a blockchain is a natural solution, but not scalable enough.

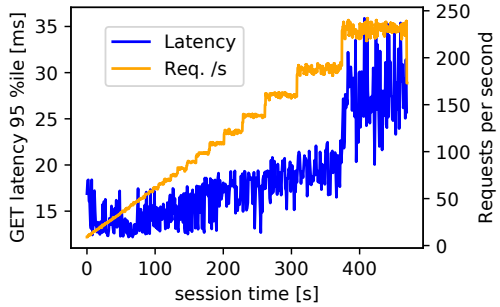
In contrast, the ULDM-based system is shared-nothing and horizontally scalable: the performance of one node does not limit the performance of the overall system. For completeness, we include single-system performance metrics here. Table 3 shows the average latency of the ULDM storage performing single 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 backed by MySQL. Fig. 7 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 [20] 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



(a) 95th %ile PUT



(b) 95th %ile GET

Figure 7: Latencies for ULDM PUT/GET as the throughput is ramped up to the single-node maximum.

possible on a blockchain system [22]. 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 [41].

8.4 Deployment Experiences

WAVE is a real-world artifact and is open source [7]. We operated various versions of WAVE for roughly two years in over 20 buildings, controlling more than 800 devices (thermostats, control processes, motion sensors, and others with little to no existing authorization capabilities) comprising 363 entities, 27 namespaces and 529 attestations (both valid and expired). The global authorization graph in our deployment is visualized in Fig. 8. The median number of delegations in a path is 4 (the maximum is 9). This deployment has given us the opportunity to refine and validate the performance, usability, and expressiveness of WAVE's authorization model in practice. Applying WAVE to legacy devices whose firmware cannot be modified is done by using an adaptation layer microservice and ensuring all communication with the legacy device flows through that service [8].

Performance. In the deployment, most proofs build in under 20ms and validate in under 10ms (as in Fig. 6b). The performance impact of WAVE is imperceptible during normal operation: proofs are cached after processing, accelerating subsequent generation and validation. As mentioned, we built an earlier version of WAVE on top of a blockchain instead of our current ULDM. We conducted extensive bench-

Work	Transitive delegation	Discoverability	No order constraints	Offline participants	No trusted central storage	Protected permissions
Auth. languages [12, 49, 13, 43, 45, 29, 27]	Yes	**No**	**Unknown: no mechanism given**			
Hidden credentials [58, 37, 30, 48]	Yes	**No**	**Unknown: no mechanism given**			
Centralized authorization [19, 15, 56, 28]	Yes	Yes	Yes	Yes	**No**	**No**
Distributed authorization [44, 59, 18, 57, 50]	Yes	Yes	Yes	**No**	Yes	**No**
WAVE	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**	**Yes**

Table 4: Related work on decentralized authorization compared to WAVE. We elaborate on these categories in §9.

marks of that version and concluded that it cannot scale past a load roughly equivalent to a city (≈ 1 million buildings). It also incurs significant CPU and bandwidth costs, even when only storing permissions (not data).

Usability. In addition to our experience with the deployment, we have also held multiple tutorials with 200+ users. User feedback indicated that WAVE improved most aspects of management (especially administrators having autonomy to grant and revoke permissions). Some aspects of WAVE are harder to manage: no user can enumerate all delegations in the system, which reduces auditability. We were able to mitigate unfamiliarity with WAVE's authorization model with careful user interface design (which provides secure defaults such as short expiry times) and with teaching users through familiar analogies (e.g., comparing RTree to file paths).

Expressiveness. We found that WAVE was able to capture exactly the authorization patterns required in typical cyber-physical usage scenarios. The transitive delegation capability was invaluable in lowering the administrative overhead of deployments. Rather than requiring the building manager to be a part of every commissioning workflow (to create credentials for each new device), permission is granted to the person heading the deployment effort, who then acts with autonomy. For permanent installations, the installing entity can be removed from the permission flow afterwards by granting "around" them directly from the building manager to the devices. For temporary installations, keeping the installing entity in the flow simplifies revocation when the study is over.

Figure 8: The permission graph for the multi-building deployment. "Bolded" nodes are namespace authorities. Most nodes with a high degree are entities that administer a set of namespaces. Leaf nodes correspond to devices and services that do not perform any delegation.

9 Related Work

Table 4, compares prior authorization and trust management systems with WAVE. Here, we provide additional details.

9.1 Trust Management and Authorization

Trust Management (TM) literature over the past two decades has thoroughly researched techniques for transitively delegable authorization. Overviews of TM systems are provided in [14, 51, 11, 6].

Languages used to express authorization policies are summarized in the first row of Table 4 [12, 49, 13, 43, 10, 27]. For example Macaroons [12] provides a mechanism for expressing authorization policy with delegation and context-specific third-party caveats. The goals are quite different, e.g. 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 [24], but mechanisms like third-party caveats could be introduced with no changes to the underlying layers.

Hidden credentials (row 2 in Table 4) [58, 37, 30, 48] address a different privacy problem: allowing a prover and verifier to hide their credentials from each other. WAVE solves an orthogonal problem: the privacy of credentials in storage and during discovery.

The remaining literature can be categorized as relying on a centralized credential store for discovery [19, 15, 56], or a distributed credential store [44, 59, 18, 57, 50]. 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 [55, 26] avoids this problem but does not scale, and

thus far has focused on identity, not authorization. Work such as [28] 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 any credential holder is offline. Both the centralized and decentralized credential discovery work thus far have overlooked the privacy of credentials at rest (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.

A concurrent work, Droplet [52], presents a distributed authorization system, but it does not meet the requirements of a general purpose authorization system in §1: Droplet does not provide transitive delegation, it only handles authorization for time series data streams as opposed to the more general policies of WAVE, and it induces a blockchain transaction for every change to an ACL, which scales poorly.

WAVEs attestations and RDE can be used as the key exchange protocol for an end-to-end encryption scheme such as JEDI [38]. JEDI provides resource-oriented message encryption on a tree of resources, which interfaces well with WAVEs RTree authorization policy.

9.2 Storage

WAVE's Map Log Root is similar to the approach used by CONIKS [47] and Key Transparency (KT) [32]. There are 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. 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 [40] 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, which requires the client/auditor to request a consistency proof between two versions of the map without knowing the contents (as it cannot construct a replica). We are not aware of any Merkle tree map databases that support this operation. A ULDM is built on simpler operations and can be constructed using an off-the-shelf database, such as Trillian [33], with full auditability.

10 Conclusion

WAVE is a *decentralized* authorization framework leveraging an improved graph-based authorization model. It introduces an encryption technique, RDE, for hiding attestation contents, while still allowing efficient discovery of permissions granted out of order to offline participants. WAVE introduces a storage mechanism, the ULDM, that is efficiently auditable. This enables untrusted, horizontally scalable, servers to store the attestations without compromising on the security of the system as a whole.

We used WAVE to manage IoT deployments in 20 buildings for two years, during which we identified six requirements that are critical for IoT deployments. In meeting these requirements, WAVE (1) has no reliance on central trust, (2) provides transitive fine-grained delegation and revocation, (3) protects permissions during discovery and at rest, (4) allows for any party to verify a proof of authorization, (5) allows delegations to occur in any order with no communication between granter and receiver, and finally (6) allows for granting permissions to offline participants. No existing work meets these requirements simultaneously. Our open-source implementation of WAVE offers similar performance to traditional centralized systems while providing stronger security guarantees.

Acknowledgements

We thank our anonymous reviewers and our shepherd for their invaluable feedback. This research was supported by Intel/NSF CPS-Security #1505773 and #20153754, DoE #DE-EE000768, NSF CISE Expeditions #CCF-1730628, NSF GRFP #DGE-1752814, and gifts from the Sloan Foundation, Hellman Fellows Fund, Alibaba, Amazon, Ant Financial, Arm, Capital One, Ericsson, Facebook, Google, Intel, Microsoft, Scotiabank, Splunk and VMware.

References

[1] Tor project: Anonymity online. `https://www.torproject.org/`.

[2] Facebook permission bug. `https://money.cnn.com/2018/06/07/technology/facebook-public-post-error/index.html`, 2018.

[3] If This Then That. `https://ifttt.com/`, 2018.

[4] OAuth 2.0. `https://oauth.net/2/`, 2018.

[5] Michel Abdalla et al. Identity-based encryption gone wild. In *ICALP*, 2006.

[6] A Ahadipour and M Schanzenbach. A survey on authorization in distributed systems: Information storage, data retrieval and trust evaluation. In *Trustcom*, 2017.

[7] Michael Andersen and Sam Kumar. Source for WAVE. `https://github.com/immesys/wave`.

[8] Michael P Andersen, John Kolb, Kaifei Chen, Gabe Fierro, David E Culler, and Randy Katz. Democratizing authority in the built environment. *TOSN*, 2018.

[9] Michael P Andersen, John Kolb, Kaifei Chen, Gabriel Fierro, David E Culler, and Raluca Ada Popa. WAVE: A decentralized authorization system for IoT via blockchain smart contracts. *UC Berkeley Tech. Rep. UCB/EECS-2017-234*, 2017.

[10] Moritz Becker et al. SecPAL: Design and semantics of a decentralized authorization language. *JCS*, 2010.

[11] Elisa Bertino, Elena Ferrari, and Anna Squicciarini. Trust negotiations: concepts, systems, and languages. *Computing in science & engineering*, 6(4), 2004.

[12] Arnar Birgisson, Joe Gibbs Politz, Ulfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *NDSS*, 2014.

[13] Matt Blaze et al. Keynote: Trust management for public-key infrastructures. In *SWP*, 1998.

[14] Matt Blaze, Joan Feigenbaum, and Jack Lacy. Decentralized trust management. In *IEEE S & P*, 1996.

[15] Matt Blaze, Joan Feigenbaum, and Martin Strauss. Compliance checking in the policymaker trust management system. In *FC*, 1998.

[16] D. Boneh and M. Franklin. Identity-based encryption from the weil pairing. In *SIAM J Comput*, 2003.

[17] Christian Cachin. Architecture of the hyperledger blockchain fabric. 2016.

[18] Ke Chen, Kai Hwang, and Gang Chen. Heuristic discovery of role-based trust chains in peer-to-peer networks. *IEEE TPDS*, 20(1):83–96, 2009.

[19] Dwaine Clarke et al. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 2001.

[20] James C Corbett et al. Spanner: Google's globally distributed database. *ACM TOCS*, 31(3):8, 2013.

[21] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. Riposte: An anonymous messaging system handling millions of users. In *IEEE S&P*, 2015.

[22] Kyle Croman et al. On scaling decentralized blockchains. In *FC*, 2016.

[23] Adam Eijdenberg, Ben Laurie, and Al Cutter. Verifiable data structures. `https://github.com/google/trillian/blob/master/docs/VerifiableDataStructures.pdf`.

[24] Carl M Ellison, Bill Frantz, Butler Lampson, Ron Rivest, Brian M Thomas, and Tatu Ylonen. SPKI examples, 1998.

[25] Ksenia Ermoshina, Francesca Musiani, and Harry Halpin. End-to-end encrypted messaging protocols: An overview. In *INRIA*, 2017.

[26] Evernym Inc. Everynm: Self-sovereign identity with verifiable claims, 2018.

[27] A. Felkner and A. Kozakiewicz. Practical extensions of trust management credentials. In *iNetSApp*. 2017.

[28] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. Decentralized action integrity for trigger-action IoT platforms. In *NDSS*, 2018.

[29] Philip WL Fong. Relationship-based access control: protection model and policy language. In *CODASPY*, 2011.

[30] Keith Frikken et al. Attribute-based access control with hidden policies and hidden credentials. *IEEE TC*, 2006.

[31] Google. GRPC, a high performance, open-source universal RPC framework. `https://grpc.io/`.

[32] Google. Key transparency. `https://github.com/google/keytransparency/blob/master/docs/design.md`.

[33] Google. Trillian. `https://github.com/google/trillian`.

[34] Google. VLBM implementation. `https://github.com/google/trillian/tree/master/examples/ct/ctmapper`.

[35] V. Goyal, O. Pandey, A. Sahai, and B. Waters. Attribute-based encryption for fine-grained access control of encrypted data. In *CCS*, 2006.

[36] OAuth Working Group. Oauth 2 token exchange. `https://tools.ietf.org/html/draft-ietf-oauth-token-exchange-15`, 2018.

[37] Jason E Holt et al. Hidden credentials. In *ACM workshop on privacy in the electronic society*, 2003.

[38] Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E. Culler. JEDI: Many-to-many end-to-end encryption and key delegation for iot. In *USENIX Security*, 2019.

[39] Selena Larson. Every single yahoo account was hacked - 3 billion in all, October 2017. Online.

[40] Ben Laurie. Revocation Transparency. `https://www.links.org/files/RevocationTransparency.pdf`, 2018.

[41] Ben Laurie, A. Langley, and E. Kasper. Certificate transparency (rfc 6992), 2013.

[42] David Lazar. Open-source IBE implementation. `https://github.com/vuvuzela/crypto`.

[43] Ninghui Li et al. Design of a role-based trust-management framework. In *IEEE S & P*, 2002.

[44] Ninghui Li et al. Distributed credential chain discovery in trust management. *J. CS, IOS Press*, 2003.

[45] Ninghui Li and John C. Mitchell. Datalog with constraints: A foundation for trust management languages. In *PADL*, 2003.

[46] Benoît Libert and Jean-Jacques Quisquater. Identity based encryption without redundancy. In *ACNS*, 2005.

[47] Marcela S. Melara et al. CONIKS: Bringing key transparency to end users. In *USENIX Security*, 2015.

[48] Sascha Müller and Stefan Katzenbeisser. Hiding the policy in cryptographic access control. In *STM*, 2011.

[49] Ronald Rivest and Butler Lampson. SDSI-a simple distributed security infrastructure. CRYPTO, 1996.

[50] Martin Schanzenbach et al. Practical decentralized attribute-based delegation using secure name systems. *arXiv:1805.06398*, 2018.

[51] Kent E. Seamons et al. Requirements for policy languages for trust negotiation. In *POLICY*. IEEE, 2002.

[52] Hossein Shafagh, Lukas Burkhalter, Simon Duquennoy, Anwar Hithnawi, and Sylvia Ratnasamy. Droplet: Decentralized authorization for iot data streams, 2018.

[53] Adi Shamir. How to share a secret. *Comm. ACM*, 1979.

[54] Mudhakar Srivatsa and Mike Hicks. Deanonymizing mobility traces: Using social network as a side-channel. In *ACM CCS*, 2012.

[55] The Sovrin Foundation. A protocol and token for self-sovereign identity and decentralized trust, 2018.

[56] Vamsi Thummala and Jeff Chase. SAFE: A declarative trust management system with linked credentials. *arXiv preprint arXiv:1510.04629*, 2015.

[57] Daniel Trivellato et al. GEM: A distributed goal evaluation algorithm for trust management. *TPLP*, 2014.

[58] Marianne Winslett, Ting Yu, Kent E Seamons, Adam Hess, Jared Jacobson, Ryan Jarvis, Bryan Smith, and Lina Yu. Negotiating trust in the web. *IEEE IC*, 2002.

[59] Xian Zhu et al. Distributed credential chain discovery in trust-management with parameterized roles. In *CANS*, 2005.

A Proof of ULDM Security Guarantee

We provide a proof sketch for Guarantee 2.

Proof Sketch for Guarantee 2. 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. □

B 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 1 (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 2 (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 WIBE 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 §4.3 and §4.4, 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 (§4.7) so we can later express the leakage of an attestation in each state.

Definition 3 (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 $A.\mathsf{subject}$ 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. 2, 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 §2.4.

Based on Definition 3, 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 3 (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 §2.1, 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 WIBE/IBE public parameters) corresponding to each entity. For entities corresponding to malicious users,* Chl *also provides the secret keys (signing key and WIBE/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 3. 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 3). 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 WIBE 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. □