WAVE: A Decentralized Authorization System for IoT via Blockchain Smart Contracts

Michael P Andersen
John Kolb
Kaifei Chen
Gabriel Fierro
David E. Culler
Raluca Ada Popa

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2017-234
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-234.html

December 29, 2017

WAVE: A Decentralized Authorization System for IoT via Blockchain Smart Contracts

Michael P Andersen, John Kolb, Kaifei Chen, Gabe Fierro, David E. Culler, Raluca Ada Popa
UC Berkeley

Abstract

Authorization is a crucial security component of many distributed systems handling sensitive data or actions, including IoT systems. We present the design of a fully decentralized authorization system, WAVE, that operates at a global scale providing fine-grained permissions, non-interactive delegation and proofs of permission that can be efficiently verified, while still supporting revocation. Using smart contracts on a public blockchain, it allows rich and complex policies to be expressed and is resistant to DoS attacks without relying on any central trusted parties. We also present a novel mechanism for protecting the secrecy of resources on the public blockchain, without out-of-band channels or interaction between granters, provers or verifiers. We implemented WAVE, which has now been running for over 500 days. We show that WAVE is efficient enough to support city-scale federation with millions of participants and permission policies.

1 Introduction

Authorization is an important security component of many distributed systems handling sensitive data or actions, whether these systems are Internet of Things (IoT) [26], Industrial Internet [27], SmartGrid [28], SmartBuildings [48], or SmartCities [18, 60]. A key aspect of these systems is that authorization permissions might span different administrative domains, each with its own internal structure [16, 19, 47].

Unfortunately, most existing authorization systems [61, 40, 45, 44, 51] rely on a trusted central authority. This is, for example, the case for many communication platforms in the IoT space (both publish-subscribe syndication and point-to-point) [6, 46, 34, 51, 38, 4, 51]. However, if an attacker compromises this party, they can subvert the authorization policy of the *entire* system. The damage of unauthorized actions represents a fundamental threat in such systems [62, 57, 22, 36].

We present WAVE *(Wide Area Verified Exchange)*, a novel authorization system spanning different administrative domains *without relying on a central authority*. Each party can express fine-grained trust into another party without implicitly having to trust a central third party. WAVE can express rich access patterns such as Role Based Access Control (RBAC) and groups. In traditional systems, the central party, when uncompromised, ensures availability, consistency, durability, and enforcement of authorization policies. In WAVE, this party is replaced with a blockchain, Ethereum [58, 12], and operations with smart contracts on blockchains [52]; this offers a distributed global, durable ledger of transactions.

The idea of using blockchains for various tasks in IoT systems has already been proposed [49, 15, 42, 3, 2, 50]. However, making this approach work for decentralized authorization in an expressive and efficient way fit for the IoT setting requires overcoming a set of challenges, which we now explain.

First, these prior systems place user or device data onto the blockchain [49, 15, 42, 3, 2, 50]. Applying this strategy to IoT systems results in critical path operations needing to perform blockchain operations, which is inefficient and expensive. Instead, WAVE uses the blockchain only for storing permissions (essentially "metadata") and not data, and does not access the blockchain upon critical data path operations, resulting in a cheaper and more efficient system.

Second, the IoT setting poses a set of new challenges: out-of-order delegations and non-interactivity. Consider, for example, that Bob grants his assistant access to everything he has access to in the "work" domain. In other words, he granted access to the *URI* "work/*". Later, an administrator installs a new lock in Bob's office and gives Bob access to it. The access from the admin to the assistant has thus been given out of order. Nevertheless, the assistant should get immediate access. Moreover, Bob might not be online ("non-interactive") at the time when the admin gave him this permission so he cannot help transfer this permission to the assistant either. WAVE addresses this problem by embedding in smart contracts cryptographically-enforced delegations, called *Delegations of Trust (DoTs)*. All the DoTs together form a *global permission graph* which spans different trust domains (such as "work" or "home"). A proof of authorization is a chain of DoTs. WAVE enables the relevant parties to look up such proofs of delegation efficiently.

Third, the global visibility of the blockchain poses privacy concerns. For example if an administrator delegates to Alice access to the resource URI "BuildingNorthGate/Floor9/Room25", an attacker essentially learns where Alice works. To protect such information on the public blockchain, a natural idea is to encrypt this information, but with whose key? Due to the out-of-order and non-interactivity properties above, one does not know ahead of time who will have access to the delegation. In fact, standard public-key encryption does not suffice here; we go through this exercise in §7.

Instead, we present a novel use of *Identity-Based Encryption (IBE)* in this setting. IBE enables a party to encrypt a message using a global public key and the identity of the receiver instead of that receiver's public key. However, to decrypt, the receiver must be granted a secret key for his identity by a global trusted entity. In WAVE, we instantiate *one IBE scheme per user*. Each user has a separate identity within a different administrative domain, called a *namespace*, such as "home" or "work". We encrypt the content of DoTs using the public IBE key of the receiving user and as identity we use the namespace, thus hiding the details of the delegation. We call the resulting DoT, a *protected DoT* or *PDoT*. Each PDoT will contain keys that enable the recipient to *recursively* decrypt PDoTs of interest on the permission graph.

Finally, by leveraging tight coupling between this authorization design and a publish-subscribe syndication system, we construct a syndication mechanism with especially useful properties in the context of IoT. Concretely, we present a publish/subscribe syndication tier that offers strong Denial Of Service resistance, maintains the decentralized and authority-free management properties present in the authorization tier, and avoids introducing implicit reliance on central authorities, such as DNS or TLS Certificate Authorities. This syndication tier does not require trusting the infrastructure that relays messages, drastically narrowing the attack surface in comparison to popular publish/subscribe systems like MQTT [6] or RabbitMQ.

We have implemented WAVE and evaluated it extensively via a set of microbenchmarks, a private blockchain fork, and a city-scale emulation to demonstrate its practicality. WAVE is open-source [5] and has been running for more than 500 days, with more than 150 IoT devices spread across our campus and homes (including thermostats, wireless sensors and Raspberry Pis), handling thousands of resources. Our private Ethereum chain is more than a year old and handles millions of emulated participants economically – WAVE can also be deployed on the main chain. This experience allowed us to tune the design of WAVE as well as identify and report issues within Ethereum, some of which have been fixed.

2 System Setup

We consider a global IoT network consisting of heterogenous devices and services at the scale of homes, buildings and cities. Devices could be thermostats, smartlocks, meters, switches, appliances, facilities, and others. Services include building controllers, city-scale electric management, and so on. Users might have smart phones, laptops, home monitors, or other devices with which they configure permissions and interact.

Designing a system for the IoT setting is challenging because this setup poses certain constraints.

C1: Heterogeneity and interaction across trust domains. There is a plethora of devices and functions they perform, some crossing trust domains. For example, Alice can have certain permissions in the context of her office and other permissions in the context of her home. She might be able to control the lights in a room, but not to unlock that room.

The following two constraints seem to be natural requirements in an IoT system, yet enforcing them cryptographically *a*t the same time is quite challenging and required new techniques in WAVE.

C2: Non-interactivity. IoT devices are intermittently connected. Hence, when granting access to an entity, we should not expect that the entity is online. When the entity comes online, it should be able to exercise its access without further interaction.

C3: Out-of-order delegation. Granting access along a chain of delegations might not happen in the order of the chain. For example, consider that Bob is giving an assistant access to everything work-related. Later, the building admin gives access to Bob to a new thermostat. The assistant should get access immediately to the thermostat. Bob essentially gave access to the thermostat before he had access to it. When combined with non-interactivity, the assistant must gain access to the thermostat *without* Bob coming online to help.

C4: Constrained resources. Many IoT devices have constrained computing, storage and energy resources.

While handling these constraints in a practical way, WAVE must also achieve these goals:

Goal 1) WAVE *must provide an integrated way to express authorization relations across trust domains and applications.* Application-specific means of expressing authorization are burdensome for the user, error-prone and make deploying applications across trust domains harder.

Goal 2) WAVE *must not rely on a central point of trust.* The IoT network can cross administrative domains with devices and services persisting beyond periods of ownership or vendor lifespan. No party can be trusted by the entire world with authentication, and any central party represents a point of attack.

Goal 3) *The system should provide protection against*

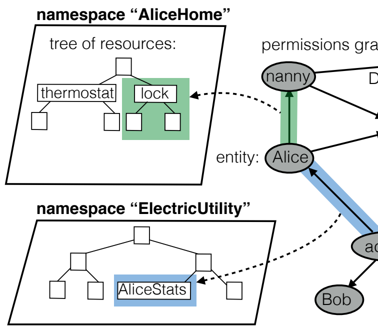

Figure 1: Overview of WAVE's authorization layer, annotated with an example explained in §3.1.

DDoS attacks. This is particularly important in light of recent high-impact attacks [41].

Goal 4) *The system should minimize privacy leakage.* With a global network, the attacker has the ability to observe actions in the network and infer private information. While removing all sources of leakage in IoT is a vastly unsolved question, one should attempt to protect the more sensitive pieces of information, such as message streams and resource hierarchies.

3 WAVE Conceptual Design

WAVE consists of three layers: an *authorization layer* which allows expression and evaluation of security policies, a *syndication layer* which provides publish/subscribe on resources and a *overlay layer* consisting of WAVE services, devices, servers and the underlying blockchain.

3.1 Authorization Layer

WAVE captures who and what has access to what using the following concepts: namespaces, resources, entities, and DoTs. Entities and DoTs form a permission graph.

Entity: An *entity* is the unit of access control for authorization. An entity is like a username or a role, except that a username/role exists only within a single domain, whereas an entity is global. Anything that grants or receives permissions is an entity and any entity can delegate permissions on its own without communicating with any other party. Anyone can create an entity.

As a running example, Alice, Bob, nanny, and admin are entities, as are the thermostat, the energy management service and the components of the infrastructure itself. An entity is a public-key pair $\langle E_{sk}, E_{vk} \rangle$, identified by its public key E_{vk}.

As it is difficult for humans to work with such keys, WAVE provides *aliases*, globally unique immutable human readable equivalents of a public key. For example, Alice's E_{vk} has alias `Alice`.

Namespace: A *namespace* is a domain containing a hierarchy of resources, e.g., Alice's home is represented by a namespace with alias `AliceHome`. A *resource* in this hierarchy can be referenced via a URI. Alice's home hierarchy in Fig. 1 corresponds to a tree with two subtrees: one for heating /cooling, including the thermostat, and one for security, including the smart lock. Each of these has child resources: 'stats' for sensor readings and 'cmd' for issuing commands (e.g., unlock). The temperature from the thermostat is at `BW://AliceHome/tstat/stats/temp`.

In traditional systems, a pre-existing "root" of permissions, such as the administrator of the authorizing server delegates permissions *within and across* namespaces. In WAVE there is no such central authority, so the entity that creates a namespace is fully authorized for all operations on all resources within that namespace. The resource URIs within a namespace begins with the E^{ns}_{vk} of the namespace entity (the root of the resource tree).

Delegation of Trust or DoT: Every other entity interacting with resources within a namespace must receive permission from the namespace entity either directly or indirectly. For instance, AliceHome is the authority for `AliceHome/*`, and ElectricUtility is for `ElectricUtility/*`. As Alice owns the home, the entity `AliceHome` grants all permissions to `Alice`.

WAVE enables distributing trust across namespaces. While Alice can access everything in her house, she cannot access anything in Bob's house, unless Bob gives her permission. Alice can delegate permissions for `lock` to Bob while he is visiting without involving the security system vendor and she can limit Bob from making further delegations.

A DoT is an edge from an entity to another entity, delegating some permission on a resource URI:

$$\text{DoT} = \langle E^{from}_{vk}, E^{to}_{vk}, \text{URI}_{rsrc}, \text{Permissions}, \text{Metadata} \rangle,$$

where the metadata is of the form $\langle$TTL, expiry time, a list of $E^{revoker}_{vk}\rangle$. The TTL allows a granter to limit further delegation. The $E^{revoker}_{vk}$ may revoke this delegation, and so can E^{from}_{sk}. Permissions in WAVE can express popular patterns such as Role Based Access Control (RBAC). To achieve this, additional entities are used to represent roles, permissions granted to that entity give permission to the role and a grant from the entity represents the ability to assume a role. To actually assume a role, the proving entity simply ensures all DoT chains used pass through the role entity. The permissions can also be simpler, such as "publish" and "subscribe" as in syndication systems. The DoT is accompanied by a signature using the secret key corresponding to E^{from}_{vk} on these entire contents, attesting that indeed E^{from}_{vk} created this DoT.

As Fig. 1 shows, each namespace has a resource tree, and there is one global permission graph. Entities are global, not tied to any namespace; and the edges between them represent DoTs. Each edge refers to resources within one namespace, but there can be multiple edges between the same two entities. For example, the entity `Alice` has some permissions in her home, and a different set of permissions at her work.

The fundamental property of WAVE is its verification of trust. An entity E_{sk}^{prover} that seeks to prove it has permissions on a resource must present a proof of authorization. The proof is constructed by finding a valid path in the DoT graph from the $E_{vk}^{namespace}$ of the resource to the E_{vk}^{prover}. The prover can construct such a proof autonomously, without communication with any of the entities involved. The proof is sent along with the details of the interaction (e.g., an RPC invocation) and recipients can efficiently verify it, also non-interactively.

A unique property of the graph is that permissions can be granted out of order, a property important for IoT systems as described in §2. An entity E_{sk}^{a} can create a DoT giving permissions to an entity E_{vk}^{b} before E_{sk}^{a} has itself received permissions. The DoT will only become useful to E_{vk}^{b} later when E_{sk}^{a} receives permissions. This ability allows revocations or key replacement to be done with minimal fuss, as downstream permissions can be 'repaired' by granting a single replacement DoT rather than having to re-issue all delegated permissions. It is also useful in device commissioning as it allows permission granting to occur in the order that suits deployment, (which is often deploying first and granting later) rather than dictating total ordering.

3.2 Syndication Layer

WAVE's authorization layer could be integrated with different syndication layers. Here, we present an example of a publish/subscribe syndication tier, which are common in IoT systems. Our contribution here is mainly the integration with the authorization layer, an offspring of which is resilience to DoS attacks.

In this syndication tier, entities subscribe to resources in the URI tree: either to a specific resource or to a subtree in the URI tree. An entity publishes to a specific URI. For example, an app on Alice's phone might subscribe to resources `BW://[..]/tstat/stat/*` whereas the physical thermostat subscribes to `BW://[..]/tstat/cmd`; when it receives a command, it performs it.

Subscribe permission means that the receiver can receive a stream of messages published to a resource. Publish permission means that the entity may publish messages to the resource. For instance, the DoT with permission 'publish' for `BW://[..]tstat/cmd/setpoint` enables setting the heating setpoint temperature.

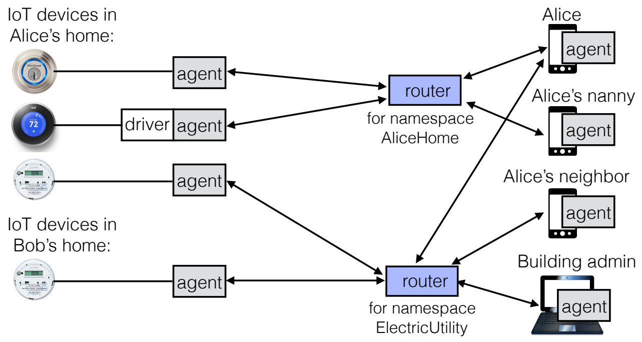

Figure 2: Example of interactions in WAVE's overlay layer. The example considers a building. The top refers to Alice's apartment and the bottom to the energy network in the building. Arrows indicate pub/sub interactions; lines between IoT devices and drivers or agents indicate a secure channel resulting from an initial pairing.

3.3 Overlay Layer

Fig. 2 illustrates the WAVE overlay comprised of *agents* and *routers*. Agents are embedded in devices, gateways, servers, or others, communicating over the WAVE protocol with routers to form the syndication overlay. In Fig. 2 the smart lock agent is embedded in the device, while that for the (legacy) smart thermostat resides as a proxy on an embedded computer in Alice's home, hosting a driver with a secure connection to the device. Agents for the electric meters might be in the device or in the utility cloud with a secure link. The Agent holds the E_{sk} for the device it represents. (Such proxy Agents typically run in a secure execution container, also represented by an E_{sk} and using WAVE authentication.)

Syndication through a namespace is enabled by binding the E_{vk}^{ns} to a *designated router*. A designated router is a node in the WAVE overlay (with a routable IP address) that *offers* to route the namespace and has its offer *accepted* by the E_{sk}^{ns}. The designated router takes responsibility for forwarding and persisting messages. Agents and routers interact with the underlying blockchain (e.g., Ethereum) through smart contracts to durably form entities, aliases, namespaces, and delegations of trust among entities on resources in namespaces, as explained in §5.

When an entity interacts with a syndication resource via its agent, the agent will form the authorization proof and attach it to a message. The router will verify the proof, dropping the message if it is invalid. The proof is a chain of valid signatures. This prevents unauthorized messages from appearing at subscribers. The subscriber also performs the canonical verification of the proof, so the router is not trusted (and cannot forge messages), but in the case that the router is honest, no bad application-layer traffic appears at the (possibly constrained) end device, simplifying firewall policy and improving DoS resilience. The router also performs revocation. It watches for revoked DoTs and no longer considers a proof relying on them as valid. The same mechanism applies to protected DoTs (explained in §7) because the signatures in the proof are provided in the clear after decryption.

4 Threat Model and Security Guarantees

We assume that the blockchain is not compromised and that each agent and router gets to see the entire blockchain or is aware that they are partitioned. This property exists in Ethereum in the absence of majority hashpower attacks. An entity or router is compromised if an attacker can see any of its secrets or affect the computation it runs in any way. All devices, servers and users (hence agents and routers) could be corrupted or malicious. They may attempt to obtain permissions they are not entitled to. An entity has legitimate access to a resource if it is granted access to that resource via a valid chain of delegations from the authority entity of the namespace containing that resource.

Authorization guarantees. The guarantee of WAVE is: *for every resource, if an attacker is not one of the entities who were legitimately given access to that resource and the attacker did not compromise any of these entities with access, the attacker cannot form a proof of authorization to that resource.*

For syndication, these guarantees hold even if the routers are compromised. If the router gets compromised, the router might send incorrect messages to subscribers to a namespace. However, WAVE ensures those subscribers will detect that the messages are unauthorized and will discard them at the agent.

DDoS protection. Assuming the designated router for a namespace is not compromised, WAVE guarantees that end hosts only receive authorized traffic. Further, the rate of creation of malicious entities, DoTs and other WAVE objects is limited because WAVE imposes time and cost on these.

Privacy guarantees. All information placed on the blockchain is public. To protect certain sensitive information, WAVE encrypts it before placing it on the blockchain. We believe WAVE makes a significant step towards protecting sensitive data. At the same time, it should be noted that privacy in blockchains and privacy in IoT systems in general is a widely unsolved problem (not particular to WAVE), and WAVE does not aim to solve all of these issues.

WAVE's PDoTs mechanism (§7) places encrypted data in the blockchain so only a small class of permitted entities (described in §7) can decrypt it. The URI is particularly important to hide; for example, the URI "building X, floor Y, office 3" in a DoT from 'admin' to 'Alice' leaks information about where Alice works and who gave her permissions. The PDoT encrypts everything except the destination entity, which is required for lookup and indexing. The only public information is "there has been some kind of delegation to Alice". We discuss in §7 which entities can decrypt this information. Additionally, note that information may leak in other ways. Some information about who gave permission to whom (the graph) might leak based on the account that made the transaction and the destination of the PDoT.

At the syndication tier, each message stream on a resource is encrypted (IND-CCA2 level), and can only be decrypted by the designated router and the entities who were legitimately given access to the resource. Unlike authorization, the designated router for a namespace is assumed to be trusted for the privacy guarantees to hold in that namespace.

WAVE does not claim to hide anything else, including metadata such as size information and timing (e.g., including the time when a transaction is made). Hiding such metadata is known to be a difficult task that often comes at high performance costs [56, 20].

Providing stronger privacy guarantees in IoT and/or blockchain-based systems is an important problem, but out of the scope of this paper.

5 WAVE Smart Contracts

The smart contract primitive [52] provided by Ethereum [12] is an immutable piece of code, executed in the Ethereum Virtual Machine (EVM) [58] by transactions to change state in the blockchain. Every node will execute the same stream of transactions, in the same order, to maintain a globally consistent view of the blockchain state, which WAVE uses to store the permission graph. Ethereum uses the notion of *gas*, a limited resource that is consumed by transactions as they execute instructions in the EVM. The maximum gas per block is decided by miner vote, imposing a global rate limit on the transactions and state created per block.

The blockchain consensus protocol is attack resistant, so WAVE inherits this resiliency by storing all global state in smart contracts. We obtain integrity by introducing the notion of a **WAVE object**, a serialized and signed version of an Entity, DoT, or revocation with associated metadata used by the four contracts below.

◇ **WAVE object contract.** This is a precompiled contract that implements a set of functions used in the other contracts: deserialize a WAVE object, take a slice from a WAVE object, compute certain hashes, or verify a signature in a WAVE object. §8.5 explains the significance of precompilation.

◇ **Registry contract.** The registry contract is used to store entities, DoTs, and revocations. It invokes functions in the WAVE object contract to check format and signature validity. Protected DoTs (§7) are opaque and not inspectable.

Registered DoT objects are indexed by E_{vk}^{to} as well as by the hash of the DoT. The index is used by entities to discover DoTs when proving authorization, discussed in §6. The contract provides accessor methods allowing constant time lookup of an entity object by E_{vk}, and a DoT object by its hash. The registry contract ensures that an entity object can only be registered once to prevent malicious modification of metadata such as expiry date.

◇ **Alias contract.** The alias contract serves to immutably bind a sequence of human readable characters to a public key. Once created they cannot be modified, which permits actions such as granting a DoT to an alias without fear that the underlying key has been replaced.

◇ **Router affinity contract.** This contract maintains a mapping from namespace entities E^{ns}_{vk} to designated router entities E^{dr}_{vk}, and a mapping from designated routers to *service records (SRV)* containing the IP address and port of the DR, to assist the syndication tier.

Out of band communication between the owner of the namespace and the owner of the designated router is used so that the designated router becomes aware of the existence of the namespace and becomes willing to route traffic for it. Once this happens, E^{dr}_{vk} creates a *Designated Routing Offer* via the router affinity contract stating that the entity belonging to the designated router is willing to route traffic for E^{ns}_{vk}.

The owner of the namespace can list all designated routing offers and, having established trust of a particular router out of band, can *accept* the designated routing offer by calling a function on the contract. After acceptance, participants interacting with resources within the namespace can resolve URIs.

Unlike the registry contract which deals with storing public objects, the affinity contract deals with actions. Ethereum guarantees that transactions are valid and signed by the originating account, but we ensure that the binding actions are authorized by the respective entities (the binding from an entity to an Ethereum account is not known by the contract). To do so, the contract maintains an incrementing nonce per E_{vk} and the actual parameters of the contract methods must include the next nonce value and be signed by the authorizing E_{vk}, preventing both spoofing and replay attacks.

6 Authorization Layer

WAVE enforces the authorization policy expressed by the permission graph without any central trusted party. Instead it leverages blockchain smart contracts (in our case, based on Ethereum).

Create entity. When creating an entity, the agent for that entity creates a WAVE object containing: $\langle$ E_{vk}, metadata, sig$\rangle$, where E_{vk} is the verification key of this entity, metadata contains information such as creation time, delegated revokers, expiry, comment, and contact, and sig is a signature using the corresponding E_{sk} on the metadata. The agent stores this object in the registry contract in the blockchain. The contract checks that the signature verifies and that the entity has not been registered before.

DoT. When creating a DoT , the agent for that entity creates a WAVE object containing $\langle E^{from}_{vk}, E^{to}_{vk}$, URI, permissions, metadata, sig$\rangle$, where sig is a signature on the entire object by E^{from}_{vk}. The agent for E^{from}_{vk} stores it in the registry. The contract checks that the specified entities are valid and that the signature is correct. The agent does not check that the granter has permissions to give this access. The designated router and subscribed agents will perform such checks when the DoT is used in a proof.

Revoke entity and DoT. Entities and DoTs can be revoked by entities in their revocation lists. If entity E^{A}_{vk} wants to revoke an object (entity/DoT), the agent of entity E^{A}_{vk} creates a signed revocation stating that the object is revoked by E^{A}_{vk} and submits it to the registry. The contract checks that E^{A}_{vk} is indeed in the list of revokers and that the request signature is valid with E^{A}_{vk}'s verification key. If an entity or DoT is expired or revoked, each participant in the blockchain will discard the corresponding entity or DoT from consideration.

7 Protected DoT

While embedding the DoT graph in the blockchain provides transparency and auditability, in some cases, this global visibility represents a privacy concern by exposing user information. For example if "admin" delegates to "Alice" a URI "building name/floor/room number", an attacker learns where Alice works. To protect such information, WAVE provides *protected DoTs* or *PDoTs*, which encrypt information in the DoT.

Recall the content of a DoT from §6. The goal of a PDoT is mainly to hide the URI from parties inspecting the blockchain, but it will encrypt other fields too. By *protected fields*, we denote the fields of a DoT that are encrypted in a PDoT. These fields are: E^{from}_{vk}, URI, Permission, TTL, and metadata including expiry date. We do not encrypt the E^{to}_{vk} because it is needed for a search of DoTs as discussed in §5 as well as determining efficiently which key to use to decrypt a PDoT (which is also important in DoS protection).

We need to encrypt a PDoT in such a way that entities in the permission graph that need to use that PDoT in a proof can decrypt it. Concretely, we require a mechanism allowing E^{A}_{vk} to grant a PDoT$_{AB}$ to E^{B}_{vk}, such that if E^{B}_{vk} grants a PDoT$_{BC}$ in the same namespace to E^{C}_{vk}, the existence of PDoT$_{BC}$ allows E^{C}_{vk} to decrypt PDoT$_{AB}$. All this must hold while satisfying the constraints of our system: out-of-order delegation and non-interactivity.

We want to limit which entities can decrypt a PDoT. Given PDoT$_{BC}$, any entity that receives access from C via any chain of PDoTs *all in the same namespace* that have the *same permission* as the PDoT (e.g., publish or subscribe), can decrypt this PDoT. No other entity, including those without a chain from C or a chain from C with different permissions, can decrypt this. We note that PDoTs can be revoked in a similar fashion to DoTs via a smart contract that points to the hash of the PDoT. However, revocation or expiry of a PDoT does not change who can decrypt that PDoT.

Insufficiency of standard cryptography. Suppose each entity, has a public and a secret key for use with PDoTs, say E^C_{vk} has $(\mathsf{SK}_C, \mathsf{PK}_C)$. Consider that whenever E^B_{vk} grants a PDoT to an entity E^C_{vk}, it encrypts the protected fields with PK_C so E^C_{vk} can decrypt it.

The problem occurs when DoTs are granted out of order, with E^B_{vk} granting a PDoT to E^C_{vk} before E^B_{vk} received a PDoT from E^A_{vk} for that resource. In this case, E^C_{vk} cannot decrypt a PDoT for E^B_{vk}, and, hence it will not have access to the resource. One possibility is for E^B_{vk} to give SK_B to E^C_{vk}. However, this means that E^C_{vk} can see all the PDoTs granted to E^B_{vk} across all namespaces E^B_{vk}. This is clearly undesirable: Bob's assistant should not see delegations in Bob's home.

To fix this, each party might have a public key per namespace; in the example above, E^B_{vk} would give a secret key for the namespace to E^C_{vk}: SK^{ns}_B. The problem here is the non-interactivity constraint (§2): an entity does not know all the namespaces it will receive access to and cannot create and announce on the relevant public keys. For example, when Bob joins a new workplace he does not know all the namespaces in the organization he will gain access to. The workspace admins would need to interact with Bob, telling him namespaces and obtaining his public key for each. Finally, the number of smart contracts and information placed on the blockchain for this task would grow with the product of entities and their relevant namespaces instead of just entities. To address this problem, we propose a novel use of *identity-based encryption*.

Identity-based encryption (IBE). In IBE [11], a master has master secret and public keys, $(\mathsf{dkm}, \mathsf{DKM})$. One can encrypt a message for a user with identity name by using the identity of the user as public key: $(\mathsf{DKM}, \mathsf{name})$. The master can create a secret key for name: $\mathsf{id}_{\mathsf{name}} \leftarrow \mathsf{IBE.Extract}(\mathsf{DKM}, \mathsf{name})$, so name can decrypt.

Design for protected DoTs. We apply IBE in a non-standard way: instead of having a trusted master and each entity being a user, we have each entity run an instance of IBE of which it is the master, and each entity has a different identity in each namespace. The "identity" of the "users" is given by the pair (namespace, permission). Using IBE, anyone can tell what is the public key of Bob in a namespace even if Bob never received a DoT in that namespace. Every E_{sk} has an IBE master private key dkm_{vk} and its corresponding IBE master public Key DKM_{vk}. The binding of E_{vk} to DKM_{vk} is made public and immutable in the registry contract. A PDoT gets registered on the blockchain in place of a regular DoT.

Consider a chain of DoTs $E^A_{vk} \Rightarrow E^B_{vk} \Rightarrow E^C_{vk} \Rightarrow \ldots$, within namespace E^{ns}_{vk}, and say we want to protect the DoT between B and C. Recall from §3.1 that the DoT consists of $G_{BC} = \langle E^C_{vk}, R\rangle$, where $R = \langle E^B_{vk}, \mathsf{URI}, E^{\mathsf{ns}}_{vk}, P, \mathsf{meta}, \mathsf{sig}\rangle$, E^{ns}_{vk} is the namespace of the resource, P is the permissions, and sig is the DoT signature. E^B_{vk} runs, at grant-time:

Protect-DoT (G_{BC}, dkm_B, DKM_C):
1. Let $\mathsf{ID}_{C,\mathsf{ns},P} \leftarrow \langle \mathsf{DKM}_C, E^{\mathsf{ns}}_{vk}, P\rangle$ be the identity of E^C_{vk} for IBE parametrized by P and E^{ns}_{vk}.
2. Let k be a freshly generated symmetric key.
3. Let $\mathsf{enck} \leftarrow \mathsf{IBE.Encrypt}(\mathsf{ID}_{C,\mathsf{ns},P}, k)$ be the wrapped symmetric key.
4. Let $\mathsf{id}_{B,\mathsf{ns},P} \leftarrow \mathsf{IBE.Extract}(\mathsf{dkm}_B, E^{\mathsf{ns}}_{vk}, P)$ be the identity private key of E^B_{vk} parameterized by P and E^{ns}_{vk}.
5. Let $R' \leftarrow \langle \mathsf{enck}, \mathsf{Encrypt}(k, \langle R, \mathsf{id}_{B,\mathsf{ns},P}\rangle)\rangle$ be the PDoT payload.
6. The PDoT is $G'_C = \langle E^C_{vk}, R'\rangle$.

Entity E^C_{vk} reveals PDoT G'_C using its IBE private key $\mathsf{id}_{C,\mathsf{ns},P}$ parameterized by P and E^{ns}_{vk}, which it can compute as $\mathsf{IBE.Extract}(\mathsf{dkm}_C, \mathsf{ID}_{C,\mathsf{ns},P})$, using:

Reveal-DoT (G'_C, $\mathsf{id}_{C,ns,P}$):
1. Parse R' from G'_C into (enck, C).
2. $k \leftarrow \mathsf{IBE.Decrypt}(\mathsf{id}_{C,\mathsf{ns},P}, \mathsf{enck})$ is the symmetric key.
3. $R, \mathsf{id}_{B,\mathsf{ns},P} \leftarrow \mathsf{Decrypt}(k, C)$.

Note that this entity can **recursively** reveal the DoT from E^A_{vk} to E^B_{vk} because the inputs are the PDoT G'_B, which it can retrieve from the blockchain, and the IBE private key $\mathsf{id}_{B,ns,P}$, which it obtained in step 3 above.

Proof verification: A proof contains the decrypted PDoTs without the IBE keys, so routers and destination agents can always verify the proof, even if they could not otherwise decrypt the PDoTs .

Proof construction: For an entity E^C_{vk} to reveal PDoTs granted to it, it applies the reveal process above recursively. Each entity has a set of decryption keys available in its local database either based on its own IBE master secret key or received from other entities during delegations. To know which key to use to decrypt, this entity tries these keys starting with those in namespaces that are more commonly used. An alternative design point is to not encrypt the namespace and the permission as part of the PDoT, but only protect the URI and other parameters, so the entity knows exactly which key to use to decrypt. However, in the real scenarios we considered, we found that this did not bring a worthy performance improvement. As a result, PDoTs are traversed in reverse compared to regular proof construction (§8.1), revealing from E^{prover}_{vk} to E^{ns}_{vk} instead of E^{ns}_{vk} to E^{prover}_{vk}. The direct order for transparent DoTs ensures that only useful DoTs are traversed – namely, DoTs that provide delegations from the namespace authority. Traversing in reverse means that E^{C}_{vk} does not know if a DoT chain is useful until it reaches the namespace authority.

However, if an agent follows this process of revealing every time it needs to show a proof, a DoS attacker might gain an advantage. While each DoT costs the attacker, the attacker has an amplification power because it can arrange a small number of DoTs in a way that creates many distinct paths.

To mitigate this, agents maintain a local cache database of *revealed* protected DoTs, containing every DoT granted on namespaces of interest, without filtering by usefulness. Proofs are now built as normal from E^{ns}_{vk} to E^{prover}_{vk} because the DoTs are decrypted locally. While this forces agents to store DoTs created by potentially unauthorized parties, this only happens once per PDoT; recall that the attacker must expend funds to create any new PDoT.

8 Overlay Layer

The overlay layer is composed of agents and routers interacting via syndication connections and via a blockchain client and its peering connections. Figure 3 gives an overview of the software components constituting a router, with additional agent components in dashed boxes.

8.1 Proof and Verification

Agents and routers work with a representation of permissions called a *DoT chain*: a proof that an entity E^{prover}_{vk} can interact with a resource URI or URI pattern on a namespace E^{ns}_{vk}. It manifests as a path through the DoT graph leading from E^{ns}_{vk} to E^{prover}_{vk}. The permissions granted by the chain are the intersection of those granted by its DoTs ; expiry or revocation of any entity or DoT in the chain invalidates the whole.

A DoT chain is serialized as a list of DoT hashes. To verify the chain a router/agent first *elaborates* the chain by resolving each hash into the full DoT object and verifying it. It then verifies that every DoT's “from” is the “to” of the previous, the intersection of the permissions is a superset of those claimed by the proof, and the TTL imposed by the DoTs in the chain is not exceeded by the length of the chain.

A *message* encapsulates an action on a resource by combining a URI, the action, a DoT hash chain terminating at E^{sender}_{vk} proving the action is permitted on the URI, and a signature by E^{sender}_{sk}. A message is verified by checking the signature and then verifying the proof.

8.2 Router Design

The structure of the router and its agent extension is illustrated in Figure 3. At the base, the **blockchain interface** layer is a modified Ethereum Go client (a variant of Geth

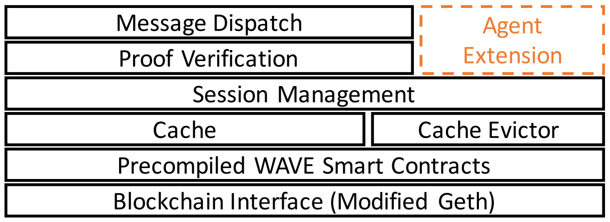

Figure 3: WAVE router/agent software design

1.5.9 [29]), complete with its active blockchain peers and blockchain state database. Above this lies an API for smart contracts. To overcome the high cost of EVM execution to extract WAVE objects from local blockchain state, a **caching layer** accelerates object resolution. To ensure the consistency of this cache, a **cache evictor** ties in to the Ethereum event log and evicts objects from the cache in response to transactions on the blockchain that might affect their validity.

Utilizing this, the **proof verification** module, given a sequence of DoT hashes, performs object resolution and verification to validate proofs.

The **message dispatch** module receives and inspects messages published to resources, passing the proof through the verification module. Invalid proofs may warrant disconnecting the remote agent and/or blacklisting the associated entities, depending on the DDOS protection policy. Messages marked as persistent are maintained in a database for asynchronous retrieval. All valid messages are matched against a resource tree formed by active subscription patterns and are inserted into matching subscription message queues.

When a remote session between a router and an agent begins, TLS with self-signed certificates is used. To avoid man-in-the-middle and masquerading attacks, the handshake concludes with both sides exchanging a signature of the TLS certificate, made with their respective E_{sk}. This allows security without using TLS certificate authority infrastructure, a centralized point of weakness [23, 30, 31].

8.3 Agent Design

The agent software shares blockchain and WAVE object caching with the router software. It differs in its connection to a device or service that it represents. Upon establishing a secure connection (hardwired in WAVE-native devices) with the agent, the device bestows its E_{sk} into the care of the agent, so that computationally intensive crypto operations can be done in the agent. IoT service logic can be implemented in any language, without the concern of having performant crypto libraries available.

The agent contains **proof generation** logic. If an agent is aware of how it has obtained permission to perform an action, it can simply concatenate DoT hashes to form the proof. To discover a proof, the agent performs a breadth first search of the permission graph searching for a path from E_{vk}^{ns} (obtained from the URI) to E_{vk}, filtering edges by the required permissions.

8.4 DDoS Protection

Routers may employ DDoS mitigation strategies, e.g. blacklisting entities that attempt too many unauthorized actions. As registering entities requires blockchain interaction, the rate at which the attacker can create new entities is severely limited, rendering an attack at this layer ineffective. See §9.1.

An attacker might attempt lower-level attacks, such as a layer 3 bandwidth DDoS. To mitigate this, we inspect the TLS handshake. Traffic categorization between valid and malicious is used to configure upstream routers to nullroute the origin IP blocks. In WAVE any session can be easily categorized as malevolent with no false positives by inspecting the first 96 bytes of the stream (containing the E_{vk}^{agent} and a signature of the TLS certificate) and comparing against the permission graph.

8.5 Precompiled Contract

While formally all four contracts could be implemented in pure Solidity [25] on the EVM, the overhead of the WAVE object contract would be excessive. Ethereum designers anticipated this and allow contracts to be *precompiled* i.e. translated into native code and bundled with the client. From the perspective of transactions and other contracts, it is a standard contract, but it consumes less CPU time and gas to execute. WAVE object verification includes signature verification on Ed25519, an elliptic curve not supported natively by the EVM.[1]

A further optimization is the persistence of state between invocations of the WAVE object contract methods. Rather than using global variables in a contract, incurring state changes in the blockchain after the end of the transaction, the WAVE precompiled contract contains a "scratch space" which allows temporary global variables for the duration of the transaction, without state in the blockchain. After an entity object has been inspected by the contract, future function calls can use the constant-size E_{vk} to refer to it, and the precompiled contract can internally elaborate it.

9 Evaluation

The overall effectiveness of WAVE has been validated through deployments at scales of hundreds of devices and thousands of resources over a period of more than 500 days. Here, we provide a systematic evaluation of the use of blockchain smart contracts as a basis for fully distributed consensus for storage, authentication, authorization, and DoS prevention. We begin with an evaluation of the lowest primitives and DDoS protection, then examine the cost for agents participating in the WAVE chain under various scenarios, and conclude with results from an emulated city-scale blockchain deployment.

To our knowledge, this is the most comprehensive and realistic evaluation of blockchain technology yet. Previous work has explored the overhead of analyzing contracts for security flaws [39] and the suitability of blockchains for data storage through short-term and relatively small-scale experiments [21]. We provide a longitudinal study of a practical blockchain application involving over 100 clients under real-world conditions.

Operation	Gas used	Approx USD
Register entity	185k	1.64
Register DoT	279k	2.47
Create specific alias	66k	0.58 + fee
Create any alias	69k	0.61
Offer routing	83k	0.79
Accept routing	53k	0.47
Revoke entity	51k	0.45 - bounty
Revoke DoT	60k	0.53 - bounty

Table 1: Gas cost of some WAVE contract operations.

Primitive	Server	Atom	R. Pi
Msg sig. gen. (2KB)	30	937	961
Msg sig. ver. (2KB)	75	1815	1524
PDoT enc	4422	290500	429976
PDoT dec w/ $\mathsf{id}_{i,ns,P}$	1503	126346	184518
PDoT dec w/ AES k	76	1649	1313

Table 2: The operation time (in microseconds) of WAVE crypto primitives

9.1 Microbenchmarks

We begin with the evaluation of interactions with the blockchain and the suite of crypto primitives.

Contract interactions An agent must expend gas to register an object. The price of gas is a miner-voted value that generally increases with load. This feedback mechanism prevents a single party from saturating the bandwidth of the blockchain. Table 1 shows the gas cost of WAVE contract interactions with the approximate USD cost.[2] When a WAVE object is registered, any ether transferred with it becomes a "revocation bounty" that is returned when the object is revoked.

Signatures - Table 2 shows the time spent verifying and signing messages on a range of platforms. Even constrained embedded systems can process thousands of signatures per second, well above the typical traffic of IoT networks.

Dispatch - The cost of transmitting and dispatching mes-

[1]In an upcoming update of Ethereum, it seems likely that Ed25519 signature verification will be supported. With this addition, we can implement WAVE object inspection in Solidity with acceptable performance, allowing WAVE to run on the main Ethereum chain.

[2]At the time of writing the exchange is 385 USD/ETH and 23 gwei/-gas

Percentile	Normal [ms]	Attack [ms]
99.9%	77.6	74.9
99%	10.4	10.7
95%	4.0	3.9
Mean	3.50	3.55

Table 3: End-to-end legitimate traffic latency under normal and attack conditions

sages in WAVE is similar to existing systems, such as MQTT, and is decoupled from the use of a blockchain for authentication. See [54, 59] for a quantification of these costs.

Proof - Building a proof requires a breadth-first search of the permission graph. As an extreme example, the Utility namespace for the meters in Table 7 governs the permissions for 360,712 meters, and contains a total of 363,636 DoTs. The meter searches these DoTs to generate a proof, finding one that traverses 3 DoTs and taking 1.4 ms in total. The proof is cached, so subsequent proof lookups complete in microseconds.

Protected DoTs The cost of the IBE primitives required for PDoTs is substantially higher than those for DoTs , as shown in the bottom half of Table 2, for the implementation from [37]. Note that these costs are paid only once for each DoT granted to an entity, and the revealed DoT is persisted for future use.

DDoS Protection To empirically confirm the effectiveness of an entity-blacklist DDoS policy at the router, we measure the impact on legitimate traffic when the designated router is under an attack comprising 130 new connections per second, each with a unique entity attempting unauthorized actions. To sustain an attack of this magnitude, an attacker needs to spend approximately 360 million gas per block registering new entities. This is two orders of magnitude larger than the gas limit of Ethereum's main chain and serves as a generous upper bound.

The legitimate traffic is a synthetic stream of roughly 1600 messages per second. We observe 10 minutes of traffic under normal conditions and 10 minutes under attack conditions (described below). The end-to-end message latency is broken down in Table 3, where we observe no detrimental effect on legitimate traffic. Moreover, the blockchain imposes a rate limit on identity creation, effectively rendering this attack vector irrelevant. This makes WAVE resistant to general Sybil attacks, a result that has not been achieved in any previous distributed authorization system.

9.2 Blockchain Characterization

A primary concern in using a blockchain is the cost of an agent participating in the chain. We perform a large scale test on over 100 WAVE nodes and emulate different levels of load on the blockchain to observe the impact on CPU, memory, and network bandwidth consumption of these nodes. We monitor the current "age" of the chain on each node. This is a measure of consistency specific to blockchain-based systems and the primary metric of fitness for purpose. If an agent cannot keep up with chain updates, it must drop messages or elevate its trust in the router, as it is potentially unaware of recent revocations.

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

Table 4: Breakdown of AMD64 cloud nodes in testbed.

During multiple experiments, we collected over 16,600 timeseries data streams, each with 10s resolution, that together contain 1.49 billion data points and cover a time period of nearly one month.

Experimental Setup We run WAVE on three different platforms. Docker containers are used to encapsulate 100 nodes, consisting of both miners and agents, on Amazon EC2. Seven `m4.16xlarge` instances feature Intel Xeon CPUs and SSD-backed storage that guarantee a baseline performance of about 1000 IOPS. We randomly distribute the containers among these hosts such that each host has at least 4 CPU cores and 8 GB of memory per container. `netem` shapes each container's network latency and bandwidth. A container is assigned to one of four networking classes based on a profile of wired Internet connections in North America [7]. The breakdown of nodes among these different constraint classes is given in Table 4. In addition to the AMD64 cloud nodes, three agents are run on Raspberry Pis with quad-core armv7l CPUs and 1 GB of RAM and three agents are run on i386 machines with Intel Atom CPUs and 2 GB of RAM.

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

1. *Fast Sync* [53] occurs when a new node is brought online. The node must synchronize with the blockchain to reach a consensus on WAVE's current state.
2. *Idle* is a period of minimal blockchain activity. Although blocks contain no transactions, new blocks are mined. Agents must process these new blocks to remain synchronized with the chain.
3. *Attack* is a period of intense activity, effectively at the level of a concerted attack on the chain by a party with infinite funds. This is compounded by WAVE's blocks becoming temporarily enlarged, accommodating more than 130 registrations, meaning each block consumes 25M gas, an order of magnitude more than the gas limit of Ethereum's main chain (roughly 4M).

Type	Idle (1h)	Normal (30h)	Attack (4h)
M,10,20,x64	3.80±0.72	3.96±0.26	4.84±0.75
A,45,20,x64	0.04±0.01	0.03±0.01	0.59±0.30
A,45,2,x64	0.05±0.01	0.03±0.01	0.48±0.32
A,1,20,armv7	0.46±0.08		1.39±0.10
A,2,2,armv7	0.15±0.01		1.58±0.11
A,1,20,atom	1.22±0.14		1.37±0.09
A,1,2,atom	0.47±0.06		0.61±0.10

Table 5: CPU utilization as [number of cores] for blockchain participation. The first column indicates the role (Agent or Miner), quantity, and peer count and architecture for each node type.

4. *Normal* is a period of typical city-scale load on the blockchain. The generation of this load is described in Section 9.3.

Table 5 summarizes the **CPU** measurements collected for all platforms. No WAVE node was CPU-bound at any point. Miners use multiple CPU cores regardless of the load on the chain, as expected. CPU consumption for regular agents was higher when the blockchain was under attack, but never exceeded ~½ a core.

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

The **bandwidth** differences across Internet speeds are not statistically significant when the agent is caught up on the chain. UDP bandwidth used for peer discovery is not significant compared to the bandwidth used for transferring state and is omitted. Table 6 shows the average bandwidth used by agents at different levels of chain activity. The bandwidth used to participate in the chain is reasonable even during periods of excessive load.

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

Chain age is the number of blocks separating the head of the blockchain from the latest block known to a node. Nodes stay up to date on the chain during idle and normal period, independent of peer count and available network

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

Table 6: Bandwidth [KiB/s] for blockchain participation. The first column indicates the role (Agent or Miner), quantity, and peer count for each node type.

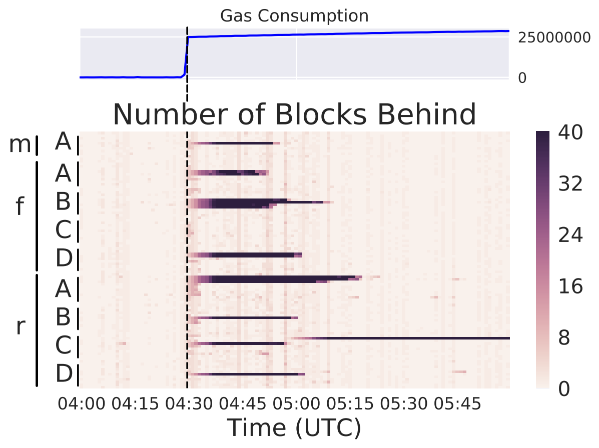

Figure 4: Number of blocks behind the head of the chain over time (x) for each node. The y axis breaks nodes down by role: miner (m), agent with a full set of 20 peers (f), and agent restricted to two peers (r) as well as by the net classes defined in Table 4.

bandwidth. We never observe a node falling behind by more than nine blocks, equivalent to maximum staleness of about two minutes.

Figure 4 shows the age of the chain for each of the 100 EC2-based WAVE nodes across our emulated attack. All nodes are up to the head of the chain until activity dramatically increases at 04:30 UTC, when most nodes fall one or two blocks behind, but quickly recover. A handful fall more behind and require about 30 minutes to recover. Upon further investigation, these nodes were found to be running on the same EC2 host, and we concluded that the instance was suffering from disk IOPS saturation.[3] 45 minutes into the attack, however, nearly all agents are again caught up to the head of the blockchain.

9.3 City-Scale Simulation

To evaluate the costs of running and participating in a blockchain-based system at real scale, we emulate a deployment of WAVE as the communication fabric for

[3]We plan to address this by provisioning more capable storage before running a new round of experiments that includes measurement of IOPS for the final version of the paper.

Type	Entities	DoTs granted	Avg Out°
Occupant	951,293	1,312,005	1.38
Apt Owner	15,787	529,562	33.54
Apt Bldg	40,921	40,921	1
Apt Lease	264,781	264,781	1
House Title	95,931	95,931	1
Thermostat	360,712	N/A	N/A
Meter	360,712	N/A	N/A
Utility	603	722,026	1197.39
Total	2,090,740	2,965,226	1.42

Table 7: Breakdown of types of entities participating in the city-scale emulation: the number of DoTs granted by entities of that type, and the average out-degree for an entity calculated as a ratio of the two counts.

smart, city-wide energy management services. The full details of the city-scale emulation are given in [1]: we summarize the scale of this emulation in Table 7. Entities represent people (e.g. homeowners), devices, and institutions (e.g. the electric utility). DoTs are granted to give building occupants the ability to actuate local thermostats, to allow the electric utility to monitor energy consumption, etc. Only control operations, such as DoT grants, require a transaction on the blockchain. Data traffic, such as meter readings, does not involve the blockchain and scales arbitrarily.

To estimate the expected load on the blockchain at this scale, we construct a statistical model of property turnover for the set of apartments, houses and occupants in San Francisco using publicly available land-use and tenancy data [17, 55]. For each day, the statistical model determines which properties in San Francisco experience turnover, generates the corresponding set of blockchain transactions, and then distributes these transactions over work hours. Each turnover event corresponds to a set of grants and revocations of DoTs and entities.

All WAVE objects were registered before the steady-state emulation. Proof construction for interacting with devices requires $\sim$1ms for a thermostat on a home/apartment namespace and $\sim$2ms for a meter on the utility namespace. Both assume a *cold cache*, so they are essentially a one-time cost. The proofs are cached in their entirety, so subsequent resolutions complete in a few microseconds. The transaction load created by this emulation remains below 150 transactions per hour (0.6 transactions per block), suggesting that even at city-scale the underlying blockchain will remain nearly idle.

10 Related Work

WAVE can be contrasted against prior frameworks for authentication and authorization along the dimensions of trust, privacy and protection.

10.1 Centralized Trust

Existing authentication and authorization systems exist in both single-authority (LDAP [61], Kerberos [40], X.509 PKI [10]) and federated (XMPP [45], Jabber [44], OAuth [33], EventGuard [51]) contexts. These rely upon a *trusted, online provider* to perform authentication, authentication and revocation services. This is the case for many platforms offering communication in the IoT space (both pub-sub syndication and point-to-point) [6, 46, 34, 51, 38, 4, 51]. WAVE can operate at large scale over untrusted infrastructure and across multiple administrative domains without the need for a third-party authenticating service.

10.2 Decentralized Trust

Several systems avoid a central authority for establishing and confirming trust relationships, authentication and authorization. We distinguish these systems by whether or not they depend upon trusted infrastructure.

CCN [35] and the Web of Trust [14, 13] implement a decentralized peer-to-peer trust model in which a principal, identified by a public key, can publish a signature of another public key to indicate a notion of trust. Chains of these signatures express delegation of trust, and are similar to DoT chains in WAVE, but cannot express *what* has been entrusted along the chain.

Trust management systems, such as PolicyMaker [9], KeyNote [8] and SPKI/SDSI [43, 24], provide more structured means of expressing trust. They combine structured policy definitions with public key identities which can be used to implement patterns such as RBAC and groups along with expiration and revocation policies. Trust relationships are established in a peer-to-peer manner and leverage signatures to verify the exchanged identities and policies.

While the *definitions* of trust and identity are fully independent from any central authority, these systems cannot establish a consistent and persistent view of the current set and status of those definitions. For example, a Web of Trust keyserver cannot prove the nonexistence of a revocation or guarantee that a key will be stored forever. WAVE resolves this by using the blockchain as a global ledger for all registered entities, DoTs and revocations, guaranteeing that all participants know the current state of all permissions (or know that they do not know).

Several recent authentication systems [2, 32, 50] leverage a blockchain to provide consistent storage for decentralized identity management services, but do not support out-of-order and non-interactive delegation and revocation as WAVE does. WAVE provides even stronger guarantees: the PDoT protocol makes private the declaration of *what* is trusted to all but the trusted parties, while still maintaining the property that these declarations can be made of out-of-order and non-interactively.

11 Conclusion

We described WAVE, a decentralized authorization system without the use of a central trusted party. WAVE leverages blockchain smart contracts combined with protected DoTs to maintain the secrecy of the resources delegated on the public blockchain. This approach provides a powerful means of federating networks of embedded networks and supporting the life cycles of devices, services, smart environments, infrastructures, and individuals. Our evaluation shows that WAVE is efficient enough to support a city-scale federation.

References

[1] Anonymized for review.

[2] Namecoin. `https://web.archive.org/web/20170830233235/https://namecoin.org/`, 2017.

[3] TILE DATA PROCESSING INC. tilepay. `https://web.archive.org/web/20170420234901/http://www.tilepay.org/`, 2017.

[4] ALLIANCE, A. AllJoyn: proximity based peer-to-peer technology. `https://web.archive.org/web/20170205050111/https://allseenalliance.org/framework`, 2017.

[5] ANONYMIZED. Open source code for WAVE. Anonymized.

[6] BANKS, A., AND GUPTA, R. Mqtt version 3.1. 1. *OASIS standard* (2014).

[7] BELSON, D. Akamai state of the internet connectivity report, q4 2016.

[8] BLAZE, M., FEIGENBAUM, J., AND KEROMYTIS, A. D. Keynote: Trust management for public-key infrastructures. In *International Workshop on Security Protocols* (1998), Springer, pp. 59–63.

[9] BLAZE, M., FEIGENBAUM, J., AND LACY, J. Decentralized trust management. In *Security and Privacy, 1996. Proceedings., 1996 IEEE Symposium on* (1996), IEEE, pp. 164–173.

[10] BOEYEN, S., SANTESSON, S., POLK, T., HOUSLEY, R., FARRELL, S., AND COOPER, D. Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. RFC 5280, May 2008.

[11] BONEH, D., AND FRANKLIN, M. Identity-based encryption from the weil pairing. In *SIAM J. of Computing,* (2003).

[12] BUTERIN, V. Ethereum: A next-generation smart contract and decentralized application platform. *URL https://github. com/ethereum/wiki/wiki/% 5BEnglish% 5D-White-Paper* (2014).

[13] CALLAS, J., DONNERHACKE, L., FINNEY, H., SHAW, D., AND THAYER, R. OpenPGP Message Format. RFC 4880 (Proposed Standard), Nov. 2007. Updated by RFC 5581.

[14] CARONNI, G. Walking the web of trust. In *Enabling Technologies: Infrastructure for Collaborative Enterprises, 2000.(WET ICE 2000). Proceedings. IEEE 9th International Workshops on* (2000), IEEE, pp. 153–158.

[15] CHAIN OF THINGS LIMITED. Chain of things. `https://web.archive.org/web/20170420234320/https://www.chainofthings.com/`, 2017.

[16] CISCO. Securing the Internet of Things: A Proposed Framework. `https://web.archive.org/web/20170421223155/https://www.cisco.com/c/en/us/about/security-center/secure-iot-proposed-framework.html`, 2016.

[17] CITY AND COUNTY OF SAN FRANCISCO. SF Open Data: Land Use. `https://data.sfgov.org/Housing-and-Buildings/Land-Use/us3s-fp9q/data`, 2015.

[18] CLARKE, R. Y. Smart cities and the internet of everything: The foundation for delivering next-generation citizen services. *Alexandria, VA, Tech. Rep* (2013).

[19] CONSORTIUM, I. I. Industrial Internet of Things Volume G4: Security Framework. `http://www.iiconsortium.org/pdf/IIC_PUB_G4_V1.00_PB.pdf`, 2016.

[20] CORRIGAN-GIBBS, H., BONEH, D., AND MAZIÈRES, D. Riposte: An anonymous messaging system handling millions of users. In *Security and Privacy (SP), 2015 IEEE Symposium on* (2015), IEEE, pp. 321–338.

[21] DINH, T. T. A., WANG, J., CHEN, G., LIU, R., OOI, B. C., AND TAN, K.-L. Blockbench: A framework for analyzing private blockchains. In *Proceedings of the 2017 ACM International Conference on Management of Data* (New York, NY, USA, 2017), SIGMOD '17, ACM, pp. 1085–1100.

[22] DROLET, M. Iot could be our downfall. `http://www.networkworld.com/article/3151912/internet-of-things/iot-could-be-our-downfall.html`, 2017.

[23] DURUMERIC, Z., KASTEN, J., BAILEY, M., AND HALDERMAN, J. A. Analysis of the https certificate ecosystem. In *Proceedings of the 2013 conference on Internet measurement conference* (2013), ACM, pp. 291–304.

[24] ELLISON, C. M., FRANTZ, B., LAMPSON, B., RIVEST, R., THOMAS, B., AND YLONEN, T. Spki certificate theory. RFC 2693, RFC Editor, September 1999. `http://www.rfc-editor.org/rfc/rfc2693.txt`.

[25] ETHEREUM. Solidity. `https://solidity.readthedocs.io/en/develop/`, 2017.

[26] EVANS, D. The internet of things: How the next evolution of the internet is changing everything. *CISCO white paper 1*, 2011 (2011), 1–11.

[27] EVANS, P. C., AND ANNUNZIATA, M. Industrial internet: Pushing the boundaries. *General Electric Reports* (2012).

[28] FARHANGI, H. The path of the smart grid. *IEEE power and energy magazine 8*, 1 (2010).

[29] GO-ETHEREUM AUTHORS. Geth: Golang ethereum client. `https://geth.ethereum.org/`, 2017.

[30] GOODIN, D. Already on probation, symantec issues more illegit https certificates, January 2017. `https://arstechnica.com/security/2017/01/already-on-probation-symantec-issues-more-illegit-https-certificates/`.

[31] GOODIN, D. Google takes symantec to the woodshed for mis-issuing 30,000 https certs, March 2017. `https://arstechnica.com/security/2017/03/google-takes-symantec-to-the-woodshed-for-mis-issuing-30000-https-certs/`.

[32] HARDJONO, T., AND SMITH, N. Cloud-based commissioning of constrained devices using permissioned blockchains. In *Proceedings of the 2nd ACM International Workshop on IoT Privacy, Trust, and Security* (2016), ACM, pp. 29–36.

[33] HARDT, D. The OAuth 2.0 Authorization Framework. RFC 6749 (Proposed Standard), Oct. 2012.

[34] HIVEMQ. HiveMQ MQTT. `https://web.archive.org/web/20161215235828/http://www.hivemq.com/hivemq/`, 2017.

[35] JACOBSON, V., SMETTERS, D. K., THORNTON, J. D., PLASS, M. F., BRIGGS, N. H., AND BRAYNARD, R. L. Networking named content. In *Proceedings of the 5th international conference on Emerging networking experiments and technologies* (2009), ACM, pp. 1–12.

[36] Kyriazis, D., Varvarigou, T., White, D., Rossi, A., and Cooper, J. Sustainable smart city iot applications: Heat and electricity management amp; eco-conscious cruise control for public transportation. In *2013 IEEE 14th International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)* (June 2013).

[37] Lazar, D. Open-source ibe implementation, part of the vuvuzela project. `https://github.com/vuvuzela/crypto`.

[38] Lutes, R. G., Haack, J., Katipamula, S., Monson, K., Akyol, B., Carpenter, B., and Tenney, N. Volttron: User guide, 2014.

[39] Luu, L., Chu, D.-H., Olickel, H., Saxena, P., and Hobor, A. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security* (New York, NY, USA, 2016), CCS '16, ACM, pp. 254–269.

[40] Neuman, B. C., and Ts'o, T. Kerberos: An authentication service for computer networks. *IEEE Communications magazine 32*, 9 (1994), 33–38.

[41] Newman, L. H. The botnet that broke the internet isnt going away. `www.wired.com/2016/12/botnet-broke-internet-isnt-going-away/`, 2016.

[42] Popov, S. The tangle. *Available electronically at http://iotatoken. com/IOTA Whitepaper. pdf* (2016).

[43] Rivest, R. L., and Lampson, B. Sdsi-a simple distributed security infrastructure. Crypto.

[44] Saint-Andre, P. Streaming xml with jabber/xmpp. *IEEE internet computing 9*, 5 (2005), 82–89.

[45] Saint-Andre, P. Extensible Messaging and Presence Protocol (XMPP): Instant Messaging and Presence. RFC 6121 (Proposed Standard), Mar. 2011.

[46] Services, A. W. AWS Internet of Things. `https://web.archive.org/web/20170413162447/https://aws.amazon.com/iot/`, 2017.

[47] Sicari, S., Rizzardi, A., Grieco, L. A., and Coen-Porisini, A. Security, privacy and trust in internet of things: The road ahead. *Computer Networks 76* (2015), 146–164.

[48] Siemens. Improving performance with integrated smart buildings. *Siemens White Paper* (2012).

[49] Slock.it UG. slock.it. `https://web.archive.org/web/20170420233936/https://slock.it/index.html`, 2017.

[50] Snow, P., Deery, B., Kirby, P., and Johnston, D. Factom ledger by consensus, 2015.

[51] Srivatsa, M., Liu, L., and Iyengar, A. Eventguard: A system architecture for securing publish-subscribe networks. *ACM Transactions on Computer Systems (TOCS) 29*, 4 (2011), 10.

[52] Szabo, N. Formalizing and securing relationships on public networks. *First Monday 2*, 9 (1997).

[53] Szilgyi, P. eth/Fast Synchronization Algorithm. `https://github.com/ethereum/go-ethereum/pull/1889`, 2015.

[54] Technologies, S. D. Benchmark of mqtt servers. `http://www.scalagent.com/IMG/pdf/Benchmark_MQTT_servers-v1-1.pdf`, 2015.

[55] U.S. Census Bureau. Population Estimates, American Community Survey, Census of Population and Housing, Current Population Survey. `https://www.census.gov/quickfacts/table/PST045216/0667000`, 2015.

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

[57] Wikipedia. 2016 dyn cyberattack. `https://en.wikipedia.org/wiki/2016_Dyn_cyberattack`, 2016.

[58] Wood, G. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper 151* (2014).

[59] Xiong, M., Parsons, J., Edmondson, J., Nguyen, H., and Schmidt, D. C. Evaluating the performance of publish/subscribe platforms for information management in distributed real-time and embedded systems. *omgwiki. org/dds* (2010).

[60] Zanella, A., Bui, N., Castellani, A., Vangelista, L., and Zorzi, M. Internet of things for smart cities. *IEEE Internet of Things journal 1*, 1 (2014), 22–32.

[61] Zeilenga, K. Lightweight Directory Access Protocol (LDAP): Technical Specification Road Map. RFC 4510 (Proposed Standard), June 2006.

[62] ZHAO, K., AND GE, L. A survey on the internet of things security. In *2013 Ninth International Conference on Computational Intelligence and Security* (Dec 2013), pp. 663–667.