JEDI: Many-to-Many End-to-End Encryption and Key Delegation for IoT

Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E. Culler
University of California, Berkeley

Abstract

As the Internet of Things (IoT) emerges over the next decade, developing secure communication for IoT devices is of paramount importance. Achieving end-to-end encryption for large-scale IoT systems, like smart buildings or smart cities, is challenging because multiple principals typically interact *indirectly* via intermediaries, meaning that the recipient of a message is not known in advance. This paper proposes JEDI (**J**oining **E**ncryption and **D**elegation for **I**oT), a many-to-many end-to-end encryption protocol for IoT. JEDI encrypts and signs messages end-to-end, while conforming to the decoupled communication model typical of IoT systems. JEDI's keys support expiry and fine-grained access to data, common in IoT. Furthermore, JEDI allows principals to delegate their keys, restricted in expiry or scope, to other principals, thereby granting access to data and managing access control in a scalable, distributed way. Through careful protocol design and implementation, JEDI can run across the spectrum of IoT devices, including ultra low-power deeply embedded sensors severely constrained in CPU, memory, and energy consumption. We apply JEDI to an existing IoT messaging system and demonstrate that its overhead is modest.

1 Introduction

As the Internet of Things (IoT) has emerged over the past decade, smart devices have become increasingly common. This trend is only expected to continue, with tens of billions of new IoT devices deployed over the next few years [33]. The IoT vision requires these devices to communicate to discover and use the resources and data provided by one another. Yet, these devices collect privacy-sensitive information about users. A natural step to secure privacy-sensitive data is to use *end-to-end encryption* to protect it during transit.

Existing protocols for end-to-end encryption, such as SSL/TLS and TextSecure [47], focus on *one-to-one* communication between two principals: for example, Alice sends a message to Bob over an insecure channel. Such protocols, however, appear not to be a good fit for large-scale industrial IoT systems. Such IoT systems demand *many-to-many* communication among **decoupled** senders and receivers, and require **decentralized delegation of access** to enforce which devices can communicate with which others.

We investigate existing IoT systems, which currently do not encrypt data end-to-end, to understand the requirements on an end-to-end encryption protocol like JEDI. We use *smart cities* as an example application area, and data-collecting sensors in a large organization as a concrete use case. We identify three central requirements, which we treat in turn below:

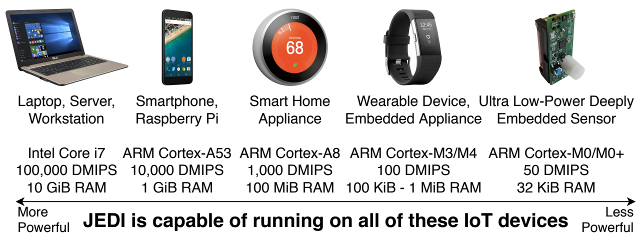


Figure 1: IoT comprises a diverse set of devices, which span more than four orders of magnitude of computing power (estimated in Dhrystone MIPS).[1]

▷ **Decoupled senders and receivers.** IoT-scale systems could consist of thousands of principals, making it infeasible for consumers of data (e.g., applications) to maintain a separate session with each producer of data (e.g., sensors). Instead, senders are typically **decoupled** from receivers. Such decoupling is common in *publish-subscribe* systems for IoT, such as MQTT, AMQP, XMPP, and Solace [81]. In particular, many-to-many communication based on publish-subscribe is the *de-facto* standard in smart buildings, used in systems like BOSS [39], VOLTTRON [87], Brume [70] and bw2 [5], and adopted commercially in AllJoyn and IoTivity. Senders publish messages by addressing them to *resources* and sending them to a *router*. Recipients *subscribe* to a resource by asking the router to send them messages addressed to that resource.

Many systems for smart buildings/cities, like sMAP [38], SensorAct [7], bw2 [5], VOLTTRON [87], and BAS [61], organize resources as a **hierarchy**. A resource hierarchy matches the organization of IoT devices: for instance, smart cities contain buildings, which contain floors, which contain rooms, which contain sensors, which produce streams of readings. We represent each resource—a leaf in the hierarchy—as a Uniform Resource Indicator (**URI**), which is like a file path. For example, a sensor that measures temperature and humidity might send its readings to the two URIs `buildingA/floor2/roomLHall/sensor0/temp` and `buildingA/floor2/roomLHall/sensor0/hum`. A user can subscribe to a URI prefix, such as `buildingA/floor2/roomLHall/*`, which represents a subtree of the hierarchy. He would then receive all sensor readings in room "LHall."

▷ **Decentralized delegation.** Access control in IoT needs to be fine-grained. For example, if Bob has an app that needs

[1]Image credits: https://tweakers.net/pricewatch/1275475/asus-f540la-dm1201t.html, https://www.lg.com/uk/mobile-phones/lg-H791, https://www.bestbuy.com/site/nest-learning-thermostat-3rd-generation-stainless-steel/4346501.p?skuId=4346501, https://www.macys.com/shop/product/fitbit-charge-2-heart-rate-fitness-wristband?ID=2999458

access to temperature readings from a single sensor, that app should receive the decryption key for only that one URI, even if Bob has keys for the entire room. In an IoT-scale system, it is not scalable for a central authority to individually give fine-grained decryption keys to each person's devices. Moreover, as we discuss in §2, such an approach would pose increased security and privacy risks. Instead, Bob, who himself has access to readings for the entire room, should be able to delegate temperature-readings access to the app. Generally, a principal with access to a set of resources can give another principal access to a subset of those resources.

Vanadium [82] and bw2 [5] introduced *decentralized delegation* (SPKI/SDSI [34] and Macaroons [14]) in the smart buildings space. Since then, decentralized delegation has become the state-of-the-art for access control in smart buildings, especially those geared toward large-scale commercial buildings or organizations [45,55]. In these systems, a principal can access a resource if there exists a *chain* of delegations, from the owner of the resource to that principal, granting access. At each link in the chain, the extent of access may be qualified by *caveats*, which add restrictions to which resources can be accessed and when. While these systems provide delegation of permissions, they do not provide protocols for encrypting and decrypting messages end-to-end.

▷ **Resource constraints.** IoT devices vary greatly in their capabilities, as shown in Fig. 1. This includes devices constrained in CPU, memory, and energy, such as wearable devices and low-cost environmental sensors.

In smart buildings/cities, one application of interest is *indoor environmental sensing*. Sensors that measure temperature, humidity, or occupancy may be deployed in a building; such sensors are *battery-powered* and transmit readings using a *low-power* wireless network. To see ubiquitous deployment, they must cost only *tens of dollars* per unit and have *several years* of battery life. To achieve this price/power point, sensor platforms are heavily resource-constrained, with mere *kilobytes* of memory (farthest right in Fig. 1) [3,4,27,44,52,63,74]. The *power consumption* of encryption is a serious challenge, even more so than its latency on a slower CPU; the CPU and radio must be used sparingly to avoid consuming energy too quickly [59, 94]. For example, on the sensor platform used in our evaluation, an average CPU utilization of merely 5% would result in less than a year of battery life, *even if the power cost of using the transducers and network were zero*.

1.1 Overview of JEDI

This paper presents JEDI, a *many-to-many* end-to-end encryption protocol compatible with the above three requirements of IoT systems. JEDI encrypts messages end-to-end for confidentiality, signs them for integrity while preserving anonymity, and supports delegation with caveats, all while allowing senders and receivers to be decoupled via a resource hierarchy. JEDI differs from existing encryption protocols like SSL/TLS, requiring us to overcome a number of *challenges*:

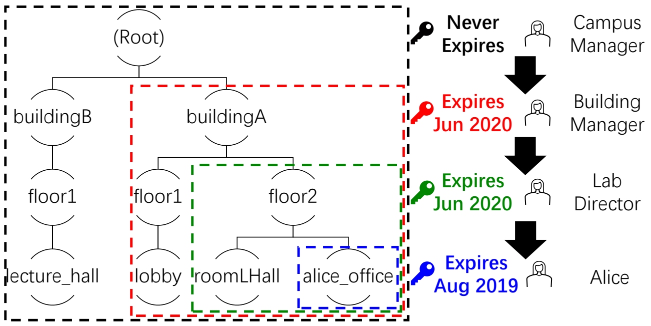


Figure 2: JEDI keys can be qualified and delegated, supporting decentralized, cryptographically-enforced access control via key delegation. Each person has a decryption key for the indicated resource subtree that is valid until the indicated expiry time. Black arrows denote delegation.

1. Formulating a new system model for end-to-end encryption to support **decoupled senders and receivers** and **decentralized delegation** typical of IoT systems (§1.1.1)
2. Realizing this expressive model while working within the **resource constraints** of IoT devices (§1.1.2)
3. Allowing receivers to verify the integrity of messages, while preserving the anonymity of senders (§1.1.3)
4. Extending JEDI's model to support revocation (§1.1.4)

Below, we explain how we address each of these challenges.

1.1.1 JEDI's System Model (§2)

Participants in JEDI are called *principals*. Any principal can create a **resource hierarchy** to represent some resources that it owns. Because that principal owns all of the resources in the hierarchy, it is called the *authority* of that hierarchy.

Due to the setting of **decoupled senders and receivers**, the sender can no longer encrypt messages with the receiver's public key, as in traditional end-to-end encryption. Instead, JEDI models principals as interacting with resources, rather than with other principals. Herein lies the key difference between JEDI's model and other end-to-end encryption protocols: the publisher of a message encrypts it according to the URI to which it is published, not the recipients subscribed to that URI. Only principals permitted to subscribe to a URI are given keys that can decrypt messages published to that URI.

IoT systems that support **decentralized delegation** (Vanadium, bw2), as well as related non-IoT authorization systems (e.g., SPKI/SDSI [34] and Macaroons [14]) provide principals with tokens (e.g., certificate chains) that they can present to prove they have access to a certain resource. Providing tokens, however, is not enough for end-to-end encryption; unlike these systems, JEDI allows *decryption keys* to be distributed via chains of delegations. Furthermore, the URI prefix and expiry time associated with each JEDI key can be restricted at each delegation. For example, as shown in Fig. 2, suppose Alice, who works in a research lab, needs access to sensor readings in her office. In the past, the campus facilities manager, who is the authority for the hierarchy, granted a key for `buildingA/*` to the building manager, who granted a key for `buildingA/floor2/*` to the lab director. Now, Alice can obtain the key for `buildingA/floor2/alice_office/*` directly from her local authority (the lab director).

1.1.2 Encryption with URIs and Expiry (§3)

JEDI supports *decoupled* communication. The resource to which a message is published acts as a *rendezvous point* between the senders and receivers, used by the underlying system to route messages. Central to JEDI is the challenge of finding an analogous *cryptographic rendezvous point* that senders can use to encrypt messages without knowledge of receivers. A number of IoT systems [75, 79] use only simple cryptography like AES, SHA2, and ECDSA, but these primitives are not expressive enough to encode JEDI's rendezvous point, which must support hierarchically-structured resources, non-interactive expiry, and decentralized delegation.

Existing systems [88–90] with similar expressivity to JEDI use Attribute-Based Encryption (ABE) [13, 51]. Unfortunately, ABE is not suitable for JEDI because it is too expensive, especially in the context of **resource constraints** of IoT devices. Some IoT systems rule it out due to its latency alone [79]. In the context of low-power devices, encryption with ABE would also consume too much power. JEDI circumvents the problem of using ABE or basic cryptography with two insights: (1) Even though ABE is too heavy for low-power devices, this does not mean that we must resort to only symmetric-key techniques. We show that certain IBE schemes [1] can be made practical for such devices. (2) **Time is another resource hierarchy**: a timestamp can be expressed as `year/month/day/hour`, and in this hierarchical representation, any time range can be represented efficiently as a logarithmic number of subtrees. With this insight, we can simultaneously support URIs and expiry via a nonstandard use of a certain type of IBE scheme: WKD-IBE [1]. Like ABE, WKD-IBE is based on bilinear groups (pairings), but it is an order-of-magnitude less expensive than ABE as used in JEDI. To make JEDI practical on low-power devices, we design it to invoke WKD-IBE *rarely*, while relying on AES most of the time, much like session keys. Thus, JEDI achieves expressivity commensurate to IoT systems that do not encrypt data—significantly more expressive than AES-only solutions—while allowing several years of battery life for low-power low-cost IoT devices.

1.1.3 Integrity and Anonymity (§4)

In addition to being encrypted, messages should be signed so that the recipient of a message can be sure it was not sent by an attacker. This can be achieved via a certificate chain, as in SPKI/SDSI or bw2. Certificates can be distributed in a decentralized manner, just like encryption keys in Fig. 2.

Certificate chains, however, are insufficient if anonymity is required. For example, consider an office space with an occupancy sensor in each office, each publishing to the same URI `buildingA/occupancy`. In aggregate, the occupancy sensors could be useful to inform, e.g., heating/cooling in the building, but individually, the readings for each room could be considered privacy-sensitive. The occupancy sensors in different rooms could use different certificate chains, if they were authorized/installed by different people. This could be used to deanonymize occupancy readings. To address this challenge, we adapt the WKD-IBE scheme that we use for end-to-end encryption to achieve an *anonymous* signature scheme that can encode the URI and expiry and support decentralized delegation. Using this technique, anonymous signatures are practical even on low-power embedded IoT devices.

1.1.4 Revocation (§5)

As stated above, JEDI keys support expiry. Therefore, it is possible to achieve a lightweight revocation scheme by delegating each key with short expiry and periodically renewing it to extend the expiry. To revoke a key, one simply does not renew it. We expect this expiry-based revocation to be sufficient for most use cases, especially for low-power devices, which typically just "sense and send."

Enforcing revocation cryptographically, without relying on expiration, is challenging. As we discuss in §5, any cryptographically-enforced scheme that provides immediate revocation (i.e., keys can be revoked without waiting for them to expire) is subject to the fundamental limitation that the sender of a message must know which recipients are revoked when it encrypts the message. JEDI provides a form of immediate revocation, subject to this constraint. We use techniques from tree-based broadcast encryption [40, 72] to encrypt in such a way that all decryption keys for that URI, *except for ones on a revocation list*, can be used to decrypt. Achieving this is nontrivial because we have to combine broadcast encryption with JEDI's semantics of hierarchical resources, expiry, and delegation. First, we modify broadcast encryption to support delegation, in such a way that if a key is revoked, all delegations made with that key are also implicitly revoked. Then, we integrate broadcast revocation, in a *non-black-box* way, with JEDI's encryption and delegation, as a third resource hierarchy alongside URIs and expiry.

1.2 Summary of Evaluation

For our evaluation, we use JEDI to encrypt messages transmitted over bw2 [5, 28], a deployed open-source messaging system for smart buildings, and demonstrate that JEDI's overhead is small in the critical path. We also evaluate JEDI for a commercially available sensor platform called "Hamilton" [52], and show that a Hamilton-based sensor sending one sensor reading every 30 seconds would see several years of battery lifetime when sending sensor readings encrypted with JEDI. As Hamilton is among the least powerful platforms that will participate in IoT (farthest to the right in Fig. 1), this validates that JEDI is practical across the IoT spectrum.

2 JEDI's Model and Threat Model

A principal can post a message to a resource in a hierarchy by encrypting it according to the resource's URI, hierarchy's public parameters, and current time, and passing it to the un-

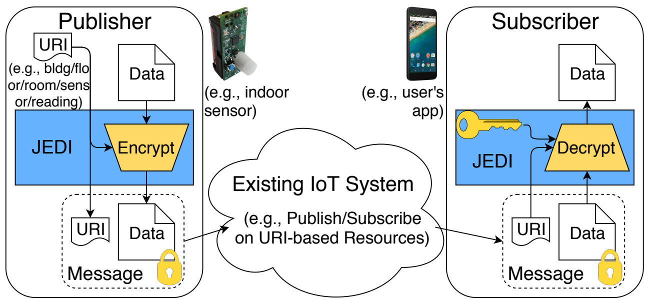


Figure 3: Applying JEDI to a smart buildings IoT system. Components introduced by JEDI are shaded. The subscriber's key is obtained via JEDI's decentralized delegation (Fig. 2).

derlying system that delivers it to the relevant subscribers. Given the secret key for a resource subtree and time range, a principal can generate a secret key for a subset of those resources and subrange of that time range, and give it to another principal, as in Fig. 2. The receiving principal can use the delegated key to decrypt messages that are posted to a resource in that subset at a time during that subrange.

JEDI does not require the structure of the resource hierarchy to be fixed in advance. In Fig. 2, the campus facilities manager, when granting access to `buildingA/*` to the building manager, need not be concerned with the structure of the subtree rooted at `buildingA`. This allows the building manager to organize `buildingA/*` independently.

2.1 Trust Assumptions

A principal is trusted for the resources it owns or was given access to (for the time ranges for which it was given access). In other words, an adversary who compromises a principal can read all resources that principal can read and forge new messages as if it were that principal. In particular, an adversary who compromises the authority for a resource hierarchy gains control over that resource hierarchy.

JEDI allows each principal to act as an authority for its own resource hierarchy in its own trust domain, without a single authority spanning all hierarchies. In particular, *principals* are not organized hierarchically; a principal may be delegated multiple keys, each belonging to a different resource hierarchy. In the example in Fig. 2, Alice might also receive JEDI keys from her landlord granting access to resources in her apartment building, in a separate hierarchy where her landlord is the authority. If Alice owns resources she would like to delegate to others, she can set up her own hierarchy to represent those resources. Existing IoT systems with decentralized delegation, like bw2 and Vanadium, use a similar model.

2.2 Applying JEDI to an Existing System

As shown in Fig. 3, JEDI can be applied as a wrapper around existing many-to-many communication systems, including publish-subscribe systems for smart cities. The transfer of messages from producers to consumers is handled by the existing system. A common design used by such systems is to have a central broker (or router) forward messages; however, an adversary who compromises the broker can read all messages. In this context, JEDI's end-to-end encryption protects data from such an adversary. Publishers encrypt their messages with JEDI before passing them to the underlying communication system (without knowledge of who the subscribers are), and subscribers decrypt them with JEDI after receiving them from the underlying communication system (without knowledge of who the publishers are).

2.3 Comparison to a Naïve Key Server Model

To better understand the benefits of JEDI's model, consider the natural strawman of a trusted key server. This key server generates a key for every URI and time. A publisher encrypts each message for that URI with the same key. A subscriber requests this key from the trusted key server, which must first check if the subscriber is authorized to receive it. The subscriber can decrypt messages for a URI using this key, and contact the key server for a new key when the key expires. JEDI's model is better than this key server model as follows:

- *Improved security.* Unlike the trusted key server, which must always be online, the authority in JEDI can delegate qualified keys to some principals *and then go offline*, leaving these principals to qualify and delegate keys further. While the authority is offline, it is more difficult for an attacker to compromise it and easier for the authority to protect its secrets because it need only access them rarely. This reasoning is the basis of root Certificate Authorities (CAs), which access their master keys infrequently. In contrast, the trusted key server model requires a central trusted party (key server) to be online to grant/revoke access to any resource.
- *Improved privacy.* No single participant sees all delegations in JEDI. An adversary in JEDI who steals an authority's secret key can decrypt all messages for that hierarchy, but still does not learn who has access to which resource, and cannot access separate hierarchies to which the first authority has no access. In contrast, an adversary who compromises the key server learns who has access to which resource and can decrypt messages for all hierarchies.
- *Improved scalability.* In the campus IoT example above, if a building admin receives access to all sensors and all their different readings for a building, the admin must obtain a potentially very large number of keys, instead of one key for the entire building. Moreover, the campus-wide key server needs to grant decryption keys to each application owned by each employee or student at the university. Finally, the campus-wide key server must understand which delegations are allowed at lower levels in the hierarchy, requiring the entire hierarchy to be centrally administered.

2.4 IoT Gateways

Low-power wireless embedded sensors, due to power constraints, often do not use network protocols like Wi-Fi, and instead use specialized low-power protocols such as Bluetooth or IEEE 802.15.4. It is common for these devices to rely on an *application-layer gateway* to send data to computers outside of the low-power network [96]. This gateway could be in the form of a phone app (e.g., Fitbit), or in the form of a specialized border router [26, 97]. In some traditional setups, the gateway is responsible for performing encryption/authentication [75]. JEDI accepts that gateways may be necessary for Internet connectivity, but does not rely on them for security—JEDI's cryptography is lightweight enough to run directly on the low-power sensor nodes. This approach prevents the gateway from becoming a single point of attack; an attacker who compromises the gateway cannot see or forge data for any device using that gateway.

2.5 Generalizability of JEDI's Model

Since JEDI decouples senders from receivers, it has no requirements on what happens at any intermediaries (e.g., does not require messages to be forwarded from publishers to subscribers in any particular way). Thus, JEDI works even when messages are exchanged in a broadcast medium, e.g., multicast. This also means that JEDI is more broadly applicable to systems with hierarchically organized resources. For example, URIs could correspond to filepaths in a file system, or URLs in a RESTful web service.

2.6 Security Goals

JEDI's goal is to ensure that principals can only read messages from or send messages to resources they have been granted access to receive from or send to. In the context of publish-subscribe, JEDI also hides the content of messages from an adversary who controls the router.

JEDI does not attempt to hide metadata relating to the actual transfer of messages (e.g., the URIs on which messages are published, which principals are publishing or subscribing to which resources, and timing). Hiding this metadata is a complementary task to achieving delegation and end-to-end encryption in JEDI, and techniques from the secure messaging literature [31, 35, 86] will likely be applicable.

3 End-to-End Encryption in JEDI

A central question answered in this section is: How should publishers encrypt messages before passing them to the underlying system for delivery (§3.4)? As explained in §1.1.2, although ABE, the obvious choice, is too heavy for low-power devices, we identify WKD-IBE, a more lightweight identity-based encryption scheme, as sufficient to achieve JEDI's properties. The primary challenge is to encode a sufficiently expressive rendezvous point in the WKD-IBE ID (called a *pattern*) that publishers use to encrypt messages (§3.4).

3.1 Building Block: WKD-IBE

We first explain WKD-IBE [1], the encryption scheme that JEDI uses as a building block. Throughout this paper, we denote the security parameter as κ.

In WKD-IBE, messages are encrypted with *patterns*, and keys also correspond to patterns. A pattern is a list of values: $P = (\mathbb{Z}_p^* \cup \{\perp\})^\ell$. The notation $P(i)$ denotes the ith component of P, 1-indexed. A pattern P_1 *matches* a pattern P_2 if, for all $i \in [1, \ell]$, either $P_1(i) = \perp$ or $P_1(i) = P_2(i)$. In other words, if P_1 specifies a value for an index i, P_2 must match it at i. Note that the "matches" operation is not commutative; "P_1 matches P_2" does not imply "P_2 matches P_1".

We refer to a component of a pattern containing an element of $\mathbb{Z}_p^*$ as *fixed*, and to a component that contains $\perp$ as *free*. To aid our presentation, we define the following sets:

Definition 1. *For a pattern S, we define:*

$$\mathit{fixed}(S) = \{(i, S(i)) \mid S(i) \neq \perp\}$$
$$\mathit{free}(S) = \{i \mid S(i) = \perp\}$$

A key for pattern P_1 can decrypt a message encrypted with pattern P_2 if $P_1 = P_2$. Furthermore, a key for pattern P_1 can be used to derive a key for pattern P_2, as long as P_1 matches P_2. In summary, the following is the syntax for WKD-IBE.

- **Setup**$(1^\kappa, 1^\ell) \to \mathsf{Params}, \mathsf{MasterKey}$;
- **KeyDer**$(\mathsf{Params}, \mathsf{Key}_{\mathsf{Pattern_A}}, \mathsf{Pattern_B}) \to \mathsf{Key}_{\mathsf{Pattern_B}}$, derives a key for $\mathsf{Pattern_B}$, where either $\mathsf{Key}_{\mathsf{Pattern_A}}$ is the $\mathsf{MasterKey}$, or $\mathsf{Pattern_A}$ matches $\mathsf{Pattern_B}$;
- **Encrypt**$(\mathsf{Params}, \mathsf{Pattern}, m) \to \mathsf{Ciphertext}_{\mathsf{Pattern}, m}$;
- **Decrypt**$(\mathsf{Key}_{\mathsf{Pattern}}, \mathsf{Ciphertext}_{\mathsf{Pattern}, m}) \to m$.

We use the WKD-IBE construction in §3.2 of [1], based on BBG HIBE [18]. Like the BBG construction, it has constant-size ciphertexts, but requires the maximum pattern length ℓ to be known at Setup time. In this WKD-IBE construction, patterns containing $\perp$ can only be used in **KeyDer**, not in **Encrypt**; we extend it to support encryption with patterns containing $\perp$. We include the WKD-IBE construction with our optimizations in Appendix A.

3.2 Concurrent Hierarchies in JEDI

WKD-IBE was originally designed to allow delegation in a *single* hierarchy. For example, the original suggested use case of WKD-IBE was to generate secret keys for a user's email addresses in all valid subdomains, such as `sysadmin@*.univ.edu` [1].

JEDI, however, uses WKD-IBE in a nonstandard way to simultaneously support *multiple* hierarchies, one for URIs and one for expiry (and later in §5, one for revocation), each in the vein of HIBE. We think of the ℓ components of a WKD-IBE pattern as "slots" that are initially empty, and are progressively filled in with calls to **KeyDer**. To combine a hierarchy of maximum depth ℓ_1 (e.g., the URI hierarchy) and a hierarchy of maximum depth ℓ_2 (e.g., the expiry hierarchy), one can **Setup** WKD-IBE with the number of slots equal to $\ell = \ell_1 + \ell_2$. The first ℓ_1 slots are filled in left-to-right for the first hierarchy and the remaining ℓ_2 slots are filled in left-to-right for the second hierarchy (Fig. 4).

3.3 Overview of Encryption in JEDI

Each principal maintains a **key store** containing WKD-IBE decryption keys. To create a resource hierarchy, any principal can call the WKD-IBE **Setup** function to create a resource hierarchy. It releases the *public parameters* and stores the *master secret key* in its key store, making it the authority of that hierarchy. To delegate access to a URI prefix for a time range, a principal (possibly the authority) searches its key store for a set of keys for a superset of those permissions. It then qualifies those keys using **KeyDer** to restrict them to the specific URI prefix and time range (§3.5), and sends the resulting keys to the recipient of the delegation.[2] The recipient accepts the delegation by adding the keys to its key store.

Before sending a message to a URI, a principal encrypts the message using WKD-IBE. The pattern used to encrypt it is derived from the URI and the current time (§3.4), which are included along with the ciphertext. When a principal receives a message, it searches its key store, using the URI and time included with the ciphertext, for a key to decrypt it.

In summary, JEDI provides the following API:

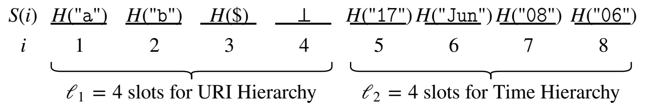
Encrypt$(\mathsf{Message}, \mathsf{URI}, \mathsf{Time}) \to \mathsf{Ciphertext}$
Decrypt$(\mathsf{Ciphertext}, \mathsf{URI}, \mathsf{Time}, \mathsf{KeyStore}) \to \mathsf{Message}$
Delegate$(\mathsf{KeyStore}, \mathsf{URIPrefix}, \mathsf{TimeRange}) \to \mathsf{KeySet}$
AcceptDelegation$(\mathsf{KeyStore}, \mathsf{KeySet}) \to \mathsf{KeyStore}'$

Note that the WKD-IBE public parameters are an implicit argument to each of these functions. Finally, although the above API lists the arguments to **Delegate** as $\mathsf{URIPrefix}$ and $\mathsf{TimeRange}$, JEDI actually supports succinct delegation over more complex sets of URIs and timestamps (see §3.7).

3.4 Expressing URI/Time as a Pattern

A message is encrypted using a pattern derived from (1) the URI to which the message is addressed, and (2) the current time. Let $H : \{0,1\}^* \to \mathbb{Z}_p^*$ be a collision-resistant hash function. Let $\ell = \ell_1 + \ell_2$ be the pattern length in the hierarchy's WKD-IBE system. We use the first ℓ_1 slots to encode the URI, and the last ℓ_2 slots to encode the time.

Given a URI of length d, such as `a/b/c` ($d = 3$ in this example), we split it up into individual components, and append a special terminator symbol `$`: `("a", "b", "c", $)`. Using H, we map each component to $\mathbb{Z}_p^*$, and then put these values into the first $d+1$ slots. If S is our pattern, we would have $S(1) = H(\texttt{"a"})$, $S(2) = H(\texttt{"b"})$, $S(3) = H(\texttt{"c"})$, and $S(4) = H(\texttt{\$})$ for this example. Now, we encode the time range into the remaining ℓ_2 slots. Any timestamp, with the granularity of an hour, can be represented hierarchically as `(year, month, day, hour)`. We encode this into the pattern like the URI: we hash each component, and assign them to consecutive slots. The final ℓ_2 slots encode the time, so the depth of the time hierarchy is ℓ_2. The terminator symbol `$` is not needed to encode the time, because timestamps always have exactly ℓ_2 components. For example, suppose that a principal sends a message to `a/b` on June 8, 2017 at 6 AM. The message is encrypted with the pattern in Fig. 4.



S(i)
H("a")
H("b")
H($)
⊥
H("17")
H("Jun")
H("08")
H("06")
i
1
2
3
4
5
6
7
8
ℓ1 = 4 slots for URI Hierarchy
ℓ2 = 4 slots for Time Hierarchy

Figure 4: Pattern S used to encrypt message sent to `a/b` on June 08, 2017 at 6 AM. The figure uses 8 slots for space reasons; JEDI is meant to be used with more slots (e.g., 20).

3.5 Producing a Key Set for Delegation

Now, we explain how to produce a key set corresponding to a URI prefix and time range. To express a URI prefix as a pattern, we do the same thing as we did for URIs, without the terminator symbol `$`. For example, `a/b/*` is encoded in a pattern S as $S(1) = H(\texttt{"a"})$, $S(2) = H(\texttt{"b"})$, and all other slots free. Given the private key for S, one can use WKD-IBE's **KeyDer** to fill in slots $3 \ldots \ell_1$. This allows one to generate the private key for `a/b`, `a/b/c`, etc.—any URI for which `a/b` is a prefix. To grant access to only a specific resource (a full URI, not a prefix), the `$` is included as before.

In encoding a time range into a pattern, single timestamps (e.g., granting access for an hour) are done as before. The hierarchical structure for time makes it possible to succinctly grant permission for an entire day, month, or year. For example, one may grant access for all of 2017 by filling in slot ℓ_2 with $H(\texttt{"2017"})$ and leaving the final $\ell_2 - 1$ slots, which correspond to month, day, and year, free. Therefore, to grant permission over a time range, *the number of keys granted is logarithmic in the length of the time range*. For example, to delegate access to a URI from October 29, 2014 at 10 PM until December 2, 2014 at 1 AM, the following keys need to be generated: `2014/Oct/29/23`, `2014/Oct/29/24`, `2014/Oct/30/*`, `2014/Oct/31/*`, `2014/Nov/*`, `2014/Dec/01/*`, and `2014/Dec/02/01`. The tree can be chosen differently to support longer time ranges (e.g., additional level representing decades), change the granularity of expiry (e.g., minutes instead of hours), trade off encryption time for key size (e.g., deeper/shallower tree), or use a more regular structure (e.g., binary encoding with logarithmic split). For example, our implementation uses a depth-6 tree (instead of depth-4), to be able to delegate time ranges with fewer keys.

In summary, to produce a key set for delegation, first determine which subtrees in the time hierarchy represent the time range. For each one, produce a separate pattern, and encode the time into the last ℓ_2 slots. Encode the URI prefix in the first ℓ_1 slots of each pattern. Finally, generate the keys corresponding to those patterns, using keys in the key store.

3.6 Optimizations for Low-Power Devices

On low-power embedded devices, performing a single WKD-IBE encryption consumes a significant amount of energy. Therefore, we design JEDI with optimizations to WKD-IBE.

[2] JEDI does not govern *how* the key set is transferred to the recipient, as there are existing solutions for this. One can use an existing protocol for one-to-one communication (e.g., TLS) to securely transfer the key set. Or, one can encrypt the key set with the recipient's (normal, non-WKD-IBE) public key, and place it in a common storage area.

3.6.1 Hybrid Encryption and Key Reuse

JEDI uses WKD-IBE in a hybrid encryption scheme. To encrypt a message m in JEDI, one samples a symmetric key k, and encrypts k with JEDI to produce ciphertext c_1. The pattern used for WKD-IBE encryption is chosen as in §3.4 to encode the *rendezvous point*. Then, one encrypts m using k to produce ciphertext c_2. The JEDI ciphertext is (c_1, c_2).

For subsequent messages, one reuses k and c_1; the new message is encrypted with k to produce a new c_2. One can keep reusing k and c_1 until the WKD-IBE pattern for encryption changes, which happens at the end of each hour (or other interval used for expiry). At this time, JEDI performs *key rotation* by choosing a new k, encrypting it with WKD-IBE using the new pattern, and then proceeding as before. Therefore, *most messages only incur cheap symmetric-key encryption*.

This also reduces the load on subscribers. The JEDI ciphertexts sent by a publisher during a single hour will all share the same c_1. Therefore, the subscriber can decrypt c_1 once for the first message to obtain k, and *cache* the mapping from c_1 to k to avoid expensive WKD-IBE decryptions for future messages sent during that hour.

Thus, expensive WKD-IBE operations are only performed upon key rotation, which happens *rarely*—once an hour (or other granularity chosen for expiry) for each resource.

3.6.2 Precomputation with Adjustment

Even with hybrid encryption and key reuse to perform WKD-IBE encryption rarely, WKD-IBE contributes significantly to the overall power consumption on low-power devices. Therefore, this section explores how to perform individual WKD-IBE encryptions more efficiently.

Most of the work to encrypt under a pattern S is in computing the quantity $Q_S = g_3 \cdot \prod_{(i,a_i)\in\text{fixed}(S)} h_i^{a_i}$, where g_3 and the h_i are part of the WKD-IBE public parameters. One may consider computing Q_S once, and then reusing its value when computing future encryptions under the same pattern S. Unfortunately, this alone does not improve efficiency because the pattern S used in one WKD-IBE encryption is different from the pattern T used for the next encryption.

JEDI, however, observes that S and T are similar; they match in the ℓ_1 slots corresponding to the URI, and the remaining ℓ_2 slots will correspond to adjacent leaves in the time tree. JEDI takes advantage of this by efficiently *adjusting* the precomputed value Q_S to compute Q_T as follows:

$$Q_T = Q_S \cdot \prod_{\substack{(i,b_i)\in\text{fixed}(T)\\ i\in\text{free}(S)}} h_i^{b_i} \cdot \prod_{\substack{(i,a_i)\in\text{fixed}(S)\\ i\in\text{free}(T)}} h_i^{-a_i} \cdot \prod_{\substack{(i,a_i)\in\text{fixed}(S)\\ (i,b_i)\in\text{fixed}(T)\\ a_i\neq b_i}} h_i^{b_i-a_i}$$

This requires one $\mathbb{G}_1$ exponentiation per differing slot between S and T (i.e., the Hamming distance). Because S and T usually differ in only the final slot of the time hierarchy, this will usually require one $\mathbb{G}_1$ exponentiation total, substantially faster than computing Q_T from scratch. Additional exponentiations are needed at the end of each day, month, and year, but they can be eliminated by maintaining additional precomputed values corresponding to the start of the current day, current month, and current year.

The protocol remains secure because a ciphertext is distributed identically whether it was computed from a precomputed value Q_S or via regular encryption as in Appendix A.

3.7 Extensions

Via simple extensions, JEDI can support (1) wildcards in the *middle* of a URI or time, and (2) forward secrecy. We describe these extensions in Appendix F.

3.8 Security Guarantee (Proof in Appendix E)

We formalize the security of JEDI's encryption below.

Theorem 1. *Suppose JEDI is instantiated with a Selective-ID CPA-secure [1, 17], history-independent (Appendix E) WKD-IBE scheme. Then, no probabilistic polynomial-time adversary $\mathcal{A}$ can win the following security game against a challenger $\mathcal{C}$ with non-negligible advantage:*
Initialization. *$\mathcal{A}$ selects a (URI, time) pair to attack.*
Setup. *$\mathcal{C}$ gives $\mathcal{A}$ the public parameters of the JEDI instance.*
Phase 1. *$\mathcal{A}$ can make three types of queries to $\mathcal{C}$:*
1. $\mathcal{A}$ asks $\mathcal{C}$ to create a principal; $\mathcal{C}$ returns a name in $\{0,1\}^*$, which $\mathcal{A}$ can use to refer to that principal in future queries. A special name exists for the authority.
2. $\mathcal{A}$ asks $\mathcal{C}$ for the key set of any principal; $\mathcal{C}$ gives $\mathcal{A}$ the keys that the principal has. At the time this query is made, the requested key may **not** *contain a key whose URI and time are both prefixes of the challenge (URI, time) pair.*
3. $\mathcal{A}$ asks $\mathcal{C}$ to make any principal delegate a key set of $\mathcal{A}$'s choice to another principal (specified by names in $\{0,1\}^*$).
Challenge. *When $\mathcal{A}$ chooses to end Phase 1, it sends $\mathcal{C}$ two messages, m_0 and m_1, of the same length. Then $\mathcal{C}$ chooses a random bit $b \in \{0,1\}$, encrypts m_b under the challenge (URI, time) pair, and gives $\mathcal{A}$ the ciphertext.*
Phase 2. *$\mathcal{A}$ can make additional queries as in Phase 1.*
Guess. *$\mathcal{A}$ outputs $b' \in \{0,1\}$, and wins the game if $b = b'$.*
The advantage of an adversary $\mathcal{A}$ is $\left|\Pr[\mathcal{A}\text{ wins}] - \frac{1}{2}\right|$.

We prove this theorem in Appendix E. Although we only achieve selective security in the standard model (like much prior work [1, 18]), one can achieve adaptive security if the hash function H in §3.5 is modeled as a random oracle [1]. It is sufficient for JEDI to use a CPA-secure (rather than CCA-secure) encryption scheme because JEDI messages are signed, as detailed below in §4.

4 Integrity in JEDI

To prevent an attacker from flooding the system with messages, spoofing fake data, or actuating devices without permission, JEDI must ensure that a principal can only send a message on a URI if it has permission. For example, an application subscribed to `buildingA/floor2/roomLHall/sensor0/temp` should be able to verify that the readings it is receiving are produced by `sensor0`, not an attacker. In addition to subscribers, an intermediate party (e.g., the router in a publish-subscribe system) may use this mechanism to filter out malicious traffic, without being trusted to read messages.

4.1 Starting Solution: Signature Chains

A standard solution in the existing literature, used by SPKI/SDSI [34], Vanadium [82], and bw2 [5], is to include a certificate chain with each message. Just as permission to subscribe to a resource is granted via a chain of delegations in §3, permission to publish to a resource is also granted via a chain of delegations. Whereas §3 includes WKD-IBE keys in each delegation, these integrity solutions delegate signed certificates. To send a message, a principal encrypts it (§3), signs the ciphertext, and includes a certificate chain that proves that the signing keypair is authorized for that URI and time.

4.2 Anonymous Signatures

The above solution reveals the sender's identity (via its public key) and the particular chain of delegations that gives the sender access. For some applications this is acceptable, and its auditability may even be seen as a benefit. For other applications, the sender must be able to send a message anonymously. See §1.1.3 for an example. How can we reconcile *access control* (ensuring the sender has permission) and *anonymity* (hiding who the sender is)?

4.2.1 Starting Point: WKD-IBE Signatures

Our solution is to use a signature scheme based on WKD-IBE. Abdalla et al. [1] observe that WKD-IBE can be extended to a signature scheme in the same vein as has been done for IBE [19] and HIBE [49]. To sign a message $m \in \mathbb{Z}_p^*$ with a key for pattern S, one uses **KeyDer** to fill in a slot with m, and presents the decryption key as a signature.

This is our starting point for designing anonymous signatures in JEDI. A message can be signed by first hashing it to $\mathbb{Z}_p^*$ and signing the hash as above. Just as consumers receive decryption keys via a chain of delegations (§3), publishers of data receive these signing keys via chains of delegations.

4.2.2 Anonymous Signatures in JEDI

The construction in §4.2.1 has two shortcomings. First, signatures are *large*, linear in the number of fixed slots of the pattern. Second, it is unclear if they are truly *anonymous*.

Signature size. As explained in §3 and Appendix A, we use a construction of WKD-IBE based on BBG HIBE [18]. BBG HIBE supports a property called *limited delegation* in which a secret key can be reduced in size, in exchange for limiting the depth in the hierarchy at which subkeys can be generated from it. We observe that the WKD-IBE construction also supports this feature. Because we need not support **KeyDer** for the decryption key acting as a signature, we use limited delegation to compress the signature to just two group elements.

Anonymity. The technique in §4.2.1 transforms an encryption scheme into a signature scheme, but the resulting signature scheme is not necessarily anonymous. For the particular construction of WKD-IBE that we use, however, we prove that the resulting signature scheme is indeed anonymous. Our insight is that, for this construction of WKD-IBE, keys are *history-independent* in the following sense: **KeyDer**, for a fixed Params and $\mathsf{Pattern}_\mathsf{B}$, returns a private key $\mathsf{Key}_{\mathsf{Pattern}_\mathsf{B}}$ with the *exact same distribution* regardless of $\mathsf{Key}_{\mathsf{Pattern}_\mathsf{A}}$ (see §3.1 for notation). Because signatures, as described in §4.2.1, are private keys generated with **KeyDer**, they are also history-independent; a signature for a pattern has the same distribution regardless of the key used to generate it. This is precisely the anonymity property we desire.

4.3 Optimizations for Low-Power Devices

As in §3.6.1, we must avoid computing a WKD-IBE signature for every message. A simple way to do this is to sample a digital signature keypair each hour, sign the verifying key with WKD-IBE at the beginning of the hour, and sign messages during the hour with the corresponding signing key.

Unfortunately, this may still be too expensive for low-power embedded devices because it requires a digital signature, which requires asymmetric-key cryptography, for *every* message. We can circumvent this by instead (1) choosing a *symmetric* key k every hour, (2) signing k at the start of each hour (using WKD-IBE for anonymity), and (3) using k in an *authenticated broadcast protocol* to authenticate messages sent during the hour. An authenticated broadcast protocol, like µTESLA [75], generates a MAC for each message using a key whose hash is the previous key; thus, the single signed key k allows the recipient to verify later messages, whose MACs are generated with hash preimages of k. In general, this design requires stricter time synchronization than the one based on digital signatures, as the key used to generate the MAC depends on the time at which it is sent. However, for the sense-and-send use case typical of smart buildings, sensors anyway publish messages on a fixed schedule (e.g., one sample every x seconds), allowing the key to depend only on the message index. Thus, timely message delivery is the only requirement. Our scheme differs from µTESLA because the first key (end of the hash chain) is signed using WKD-IBE.

Additionally, we use a technique similar to precomputation with adjustment (§3.6.2) for anonymous signatures. Conceptually, **KeyDer**, which is used to produce signatures, can be understood as a two-step procedure: (1) produce a key of the correct form and structure (called **NonDelegableKeyDer**), and (2) re-randomize the key so that it can be safely delegated (called **ResampleKey**). Re-randomization can be accelerated using the same precomputed value Q_S that JEDI uses for encryption (§3.6.2), which can be efficiently adjusted from one pattern to the next. The result of **NonDelegableKeyDer** can also be adjusted to obtain the corresponding result for a similar pattern more efficiently. We fully explain our adjustment technique for signatures in Appendix A.3.

Finally, WKD-IBE signatures as originally proposed (§4.2.1) are verified by encrypting a random message under the pattern corresponding to the signature, and then attempting to decrypt it using the key acting as a signature. We provide a more efficient signature verification algorithm for this construction of WKD-IBE in Appendix A.

4.4 Security Guarantee (Proof in Appendix E)

The integrity guarantees of the method in this section can be formalized using a game very similar to the one in Theorem 1, so we do not present it here for brevity. We do, however, formalize the anonymous aspect of WKD-IBE signatures:

Theorem 2. *For any well-formed keys k_1, k_2 corresponding to the same (URI, time) pair in the same resource hierarchy, and any message $m \in \mathbb{Z}_p^*$, the distribution of signatures over m produced using k_1 is information-theoretically indistinguishable from (i.e., equal to) the distribution of signatures over m produced using k_2.*

This implies that even a powerful adversary who observes the private keys held by all principals cannot distinguish signatures produced by different principals, for a fixed message and pattern. No computational assumptions are required. We prove Theorem 2 in Appendix E.

5 Revocation in JEDI

This section explains how JEDI keys may be revoked.

5.1 Simple Solution: Revocation via Expiry

A simple solution for revocation is to rely on expiration. In this solution, all keys are time-limited, and delegations are periodically refreshed, according to a higher layer protocol, by granting a new key with a later expiry time. In this setup, the principal who granted a key can easily revoke it by not refreshing that delegation when the key expires. We expect this solution to be sufficient for many applications of JEDI.

5.2 Immediate Revocation

Some disadvantages of the solution in §5.1 are that (1) principals must periodically come online to refresh delegations, and (2) revocation only takes effect when the delegated key expires. We would like a solution without these disadvantages.

However, any revocation scheme that does not wait for keys to expire is subject to set of *inherent* limitations. The recipient of the revoked delegation still has the revoked decryption key, so it can still decrypt messages encrypted in the same way. This means that we must either (1) rely on intermediate parties to modify ciphertexts so that revoked keys cannot decrypt them, or (2) require senders to be aware of the revocation, and encrypt messages in a different way so that revoked keys cannot decrypt them. Neither solution is ideal: (1) makes assumptions about how messages are delivered, which we have avoided thus far (§2), and requires trust in an intermediary to modify ciphertexts, and (2) weakens the decoupling of senders and receivers (§1.1). We adopt the second compromise: while senders will not need to know who are the receivers, they will need to know who has been revoked.

5.3 Immediate Revocation in JEDI

We extend tree-based broadcast encryption [40,72] to support decentralized delegation of decryption keys, and incorporate it into JEDI. We use tree-based broadcast encryption because it only requires senders to know about *revoked* users when encrypting messages, as opposed to *all* users in the system (as is required by other broadcast encryption schemes).

5.3.1 Tree-based Broadcast Encryption

Existing work [40,72] proposes two methods of tree-based broadcast encryption: Complete Subtree (CS) and Subset Difference (SD). We focus on the CS method here.

The CS method is based on a binary tree (Fig. 5) where each node corresponds to a separate keypair. Each user corresponds to a leaf of the tree and has the secret keys for all nodes on the root-to-leaf path. To encrypt a message that is decryptable by a subset of users, one finds a collection of subtrees that include all leaves except those corresponding to revoked users and encrypts the message multiple times using the public keys corresponding to the root of each subtree. By associating each node with an ID and encrypting with IBE, one can avoid generating a separate keypair for each node.

5.3.2 Modifying Broadcast Encryption for Delegation

Users in broadcast encryption do not map one-to-one to users in JEDI. To avoid confusion, we refer to "users" in broadcast encryption as "leaves" (abbreviated lf).

We modify the CS method to support delegation, as follows. Each key corresponds to a range of consecutive leaves. When a user qualifies a key to delegate to another principal, she produces a new key corresponding to a subrange of the leaves of the original key. When a key is revoked, publishers are informed of the range of leaves corresponding to the revoked key. Then, they encrypt new messages using the CS method, choosing subtrees that cover all leaves except those corresponding to revoked leaves. If a key is revoked, that key and all keys derived from it can no longer decrypt messages, which is a property that we want. Thus, if Alice has k leaves, she must store secret keys for $O(k+\log n)$ nodes, where n is the total number of leaves (so the depth of the tree is $\log n$).

In JEDI, we reduce this to $O(\log n)$ secret keys by using HIBE. We give each node v_i an identifier $\mathsf{id}(v_i) \in \{0,1\}^*$ that describes the path from the root of the tree to that node. In particular, if v_j is an ancestor of v_i, then $\mathsf{id}(v_j)$ is a prefix of $\mathsf{id}(v_i)$. Note that if we use HIBE with these IDs directly, a user with the secret key for the root can generate keys for all nodes in the tree. To fix this, we use a property called *limited delegation*, introduced by prior work [18], to generate a HIBE key that is unqualifiable (i.e., cannot be extended). For example, if Alice has leaves lf_3 to lf_4 in Fig. 5, she stores an unqualifiable key for node v_1 and a qualifiable key for node v_3. In general, each user must store $O(\log k)$ qualifiable keys and $O(\log n)$ unqualifiable keys, thus $O(\log k + \log n)$ total.

5.3.3 Using Delegable Broadcast Encryption in JEDI

Secret keys in our modified broadcast encryption scheme consist of HIBE keys, so incorporating it into JEDI is simple. As discussed in §3.2, JEDI uses WKD-IBE in a way that provides multiple concurrent hierarchies, each in the vein of

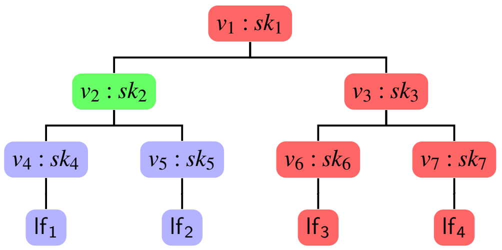


Figure 5: Key management of the CS method. Red nodes indicate nodes associated with revoked leaves. The green node is the root of the subtree covering unrevoked leaves.

HIBE. Therefore, we can instantiate a third hierarchy of depth $\ell_3 = \log n$ and use it for revocation.

Let r be the number of revoked keys. The CS method has $O(r \log \frac{n}{r})$-size ciphertexts, so JEDI ciphertexts grow to this size when revocation is used. When encrypting a message, senders use the same encryption protocol from §3 for the first $\ell_1 + \ell_2$ slots, and repeat the process, filling in the remaining ℓ_3 slots with the ID of each node used for broadcast encryption. The size of secret keys is $O(\log k + \log n)$ after our modifications to the CS method, so JEDI keys grow by this factor, to a total of $O((\log k + \log n) \cdot \log T)$ WKD-IBE keys, where T is the length of the time range for expiry.

The construction in this section works to revoke decryption keys, but cannot be used with anonymous signatures (§4.2). Extensions of tree-based broadcast encryption to signatures exist [64, 65], and we expect them to be useful to develop a construction for anonymous signatures.

How can JEDI inform publishers which leaves are revoked? One simple option is to have a global revocation list, which principals can append to. However, storing this information in a single list becomes a central point of attack, which we have avoided in our system thus far (§2). To avoid this, one can store the revocation list in a global-scale blockchain, such as Bitcoin or Ethereum, which would require an adversary to be exceptionally powerful to mount a successful attack. When revoking a set of leaves, a principal uses those keys to sign a predetermined object (as in §4.2), proving it owns an ancestor of that key in the hierarchy. To keep the revocation list private, one can use JEDI's encryption to ensure that only principals with permission to publish to a particular resource can see which keys are revoked for that resource (since publishers too have signing keys, as described in §4).

5.4 Security Guarantee (Proof in Appendix E)

The security guarantee for immediate revocation can be stated as a modification to the game in Theorem 1. In the Initialization Phase, when $\mathcal{A}$ gives $\mathcal{C}$ the challenge (URI, time), $\mathcal{A}$ additionally submits a list of revoked leaves. Furthermore, $\mathcal{A}$ may compromise principals in possession of private keys that can decrypt the challenge (URI, time) pair during Phases 1 and 2, as long as all leaves corresponding to those keys are in the revocation list submitted in the Initialization Phase. We provide a proof in Appendix E.

5.5 Optimizing JEDI's Immediate Revocation

A single JEDI ciphertext, with revocation enabled, consists of $O(r \log \frac{n}{r})$ WKD-IBE ciphertexts. To compute them efficiently, we observe that there is a large overlap in the patterns used in individual WKD-IBE encryptions, allowing us to use the "precomputation with adjustment" strategy from §3.6.2.

Even with the above optimization, immediate revocation substantially increases the cost of JEDI's cryptography. To reduce this cost, we make three observations. First, to extend JEDI's hybrid encryption to work with revocation, it is sufficient to additionally rotate keys whenever the revocation list changes, in addition to the end of each hour (as in §3.6.1). This means that, in the common case where the revocation list does not change in between two messages, efficient symmetric-key encryption can be used. Second, the revocation list used to encrypt a message need only contain revoked leaves for the *particular URI* to which the message is sent. This not only makes the broadcast encryption more efficient (smaller r), but also causes the effective revocation list for a stream of data to change even more rarely, allowing JEDI to benefit more from hybrid encryption. Third, we can do the same thing as above using the expiry time rather than the URI, allowing us to *cull* the revocation list by removing keys from it once they expire.

The efficiency of hybrid encryption depends on the revocation list changing *rarely*. We believe this is a reasonable assumption; most revocation will be handled by expiry, so immediate revocation is only needed if a principal must lose access *unexpectedly*. In the smart buildings use case (§1), for example, a key would need to be revoked if a principal unexpectedly transfers to another job.

The SD method for tree-based broadcast encryption can also be extended to support delegation and incorporated into JEDI (Appendix B), The SD method has smaller ciphertexts but larger keys.

6 Implementation

We implemented JEDI as a library in the Go programming language. We expect that only a few applications will require the anonymous signature protocol in §4.2 or the tree-based revocation protocol in §5.3; most applications can use signature chains (§4.1) for integrity and expiry for revocation (§5.1). Therefore, our implementation makes anonymous signatures optional and implements revocation separately. We expect JEDI's key delegation to be computed on relatively powerful devices, like laptops, smartphones, or Raspberry Pis; less powerful devices (e.g., right half of Fig. 1) will primarily send and receive messages, rather than generate keys for delegation. Therefore, our focus for low-power platforms was on the "sense-and-send" use case [27, 41, 44] typical of indoor environmental sensing, where a device periodically publishes sensor readings to a URI. Whereas our Go library provides higher-level abstractions, we expect low-power devices to use JEDI's crypto library directly.

6.1 C/C++ Library for JEDI's Cryptography

As part of JEDI, we implemented a cryptography library optimized in assembly for three different architectures typical of IoT platforms (Fig. 1). It implements WKD-IBE and JEDI's optimizations and modifications (in §3.6, §4.3, Appendix A). The construction of WKD-IBE is based on a bilinear group in which the Bilinear Diffie-Hellman Exponent assumption holds. We originally planned to use Barreto-Naehrig elliptic curves [32, 58] to implement WKD-IBE. Unfortunately, a recent attack on Barreto-Naehrig curves [60] reduced their estimated security level from 128 bits to at most 100 bits [11]. Therefore, we use the recent BLS12-381 elliptic curve [25].

State-of-the-art cryptography libraries implement BLS12-381, but none of them, to our knowledge, optimize for microarchitectures typical of low-power embedded platforms. To improve energy consumption, we implemented BLS12-381 in C/C++, profiled our implementation, and re-wrote performance-critical routines in assembly. We focus on ARM Cortex-M, an IoT-focused family of 32-bit microprocessors typical of contemporary low-power embedded sensor platforms [29, 52, 56]. Cortex-M processors have been used in billions of devices, including commercial IoT offerings such as Fitbit and Nest Protect. Our assembly targets Cortex-M0+, which is among the least powerful of processors in the Cortex-M series, and of those used in IoT devices (farthest to the right in Fig. 1). By demonstrating the practicality of JEDI on Cortex-M0+, we establish that JEDI is viable across the spectrum of IoT devices (Fig. 1).

The main challenge in targeting Cortex-M0+ is that the 32-bit multiply instruction provides only the lower 32 bits of the product. Even on more powerful microarchitectures without this limitation (e.g., Intel Core i7), most CPU time ($\geq 80\%$) is spent on multiply-intensive operations (e.g., BigInt multiplication and Montgomery reduction), so the lack of such an instruction was a performance bottleneck. As a workaround, our assembly code emulates multiply-accumulate with carry in 23 instructions. Cortex-M3 and Cortex-M4, which are more commonly used than Cortex-M0+, have instructions for 32-bit multiply-accumulate which produce the entire 64-bit result; we expect JEDI to be more efficient on those processors.

We also wrote assembly to optimize BLS12-381 for x86-64 and ARM64, representative of server/laptop and smartphone/Raspberry Pi, respectively (first two tiers in Fig. 1). Thus, our Go library, which runs on these non-low-power platforms, also benefits from low-level assembly optimizations.

6.2 Application of JEDI to bw2

We used our JEDI library to implement end-to-end encryption in bw2, a syndication and authorization system for IoT. bw2's syndication model is based on publish-subscribe, explained in §1. Here we discuss bw2's authorization model. Access to resources is granted via certificate chains from the authority of a resource hierarchy to a principal. Individual certificates are called Declarations of Trust (DOTs). bw2 maintains a publicly accessible registry of DOTs, implemented using blockchain smart contracts, so that principals can find the DOTs they need to form DOT chains. A *trusted* router enforces permissions granted by DOTs. Principals must present DOT chains when publishing/subscribing to resources, and the router verifies them. Note that a compromised router can read messages.

We use JEDI to enforce bw2's authorization semantics with end-to-end encryption. DOTs granting permission to subscribe now contain WKD-IBE keys to decrypt messages. By default, DOTs granting permission to publish to a URI remain unchanged, and are used as in §4.1. WKD-IBE keys may also be included in DOTs granting publish permission, for anonymous signatures (§4.2). Using our library for JEDI, we implemented a wrapper around the bw2 client library. It transparently encrypts and decrypts messages using WKD-IBE, and includes WKD-IBE parameters and keys in DOTs and principals, as needed for JEDI. bw2 signs each message with a digital signature (first alternative in §4.3).

The bw2-specific wrapper is less than 900 lines of Go code. Our implementation required no changes to bw2's client library, router, blockchain, or core—it is a separate module. Importantly, it provides the same API as the standard bw2 client library. Thus, it can be used as a drop-in replacement for the standard bw2 client library, to easily add end-to-end encryption to existing bw2 applications with minimal changes.

Table 1: Latency of JEDI's implementation of BLS12-381

Operation	**Laptop**	**Rasp. Pi**	**Sensor**
$\mathbb{G}_1$ Mul. (Chosen Scalar)	109 µs	1.33 ms	509 ms
$\mathbb{G}_2$ Mul. (Chosen Scalar)	343 µs	3.86 ms	1.44 s
$\mathbb{G}_T$ Mul. (Rand. Scalar)	504 µs	5.47 ms	1.90 s
$\mathbb{G}_T$ Mul. (Chosen Scalar)	507 µs	5.48 ms	2.81 s
Pairing	1.29 ms	14.0 ms	4.99 s

7 Evaluation

We evaluate JEDI via microbenchmarks, determine its power consumption on a low-power sensor, measure the overhead of applying it to bw2, and compare it to other systems.

7.1 Microbenchmarks

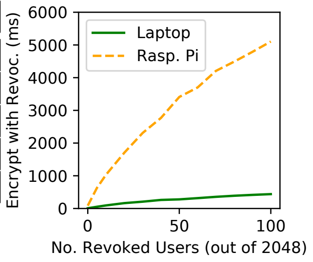
Benchmarks labeled "Laptop" were produced on a Lenovo T470p laptop with an Intel Core i7-7820HQ CPU @ 2.90 GHz. Benchmarks labeled "Raspberry Pi" were produced on a Raspberry Pi 3 Model B+ with an ARM Cortex-A53 @ 1.4 GHz. Benchmarks labeled "Sensor" were produced on a commercially available ultra low-power environmental sensor platform called "Hamilton" with an ARM Cortex-M0+ @ 48 MHz. We describe Hamilton in more detail in §7.3.

7.1.1 Performance of BLS12-381 in JEDI

Table 1 compares the performance of JEDI's BLS12-381 implementation on the three platforms, with our assembly optimizations. As expected from Fig. 1, the Raspberry Pi performance is an order of magnitude slower than Laptop performance, and performance on the Hamilton sensor is an additional two-to-three orders of magnitude slower.

	Laptop	Rasp. Pi
Enc.	3.08 ms	37.3 ms
Dec.	3.61 ms	43.9 ms
KeyD.	4.77 ms	58.5 ms
Sign	4.80 ms	61.2 ms
Verify	4.78 ms	56.3 ms

(a) Latency of **Encrypt**, **Decrypt**, **KeyDer**, **Sign**, and **Verify** with 20 attributes



(b) Encryption with Revocation

Figure 6: Performance of JEDI's cryptography

7.1.2 Performance of WKD-IBE in JEDI

Fig. 6 depicts the performance of JEDI's cryptography primitives. Fig. 6 does not include the sensor platform; §7.3 thoroughly treats performance of JEDI on low-power sensors.

In Fig. 6a, we used a pattern of length 20 for all operations, which would correspond to, e.g., a URI of length 14 and an Expiry hierarchy of depth 6. To measure decryption and signing time, we measure the time to decrypt the ciphertext or sign the message, plus the time to generate a decryption key for that pattern or ID. For example, if one receives a message on `a/b/c/d/e/f`, but has the key for `a/*`, he must generate the key for `a/b/c/d/e/f` to decrypt it.

Fig. 6a demonstrates that the JEDI encrypts and signs messages and generates qualified keys for delegation at practical speeds. On a laptop, all WKD-IBE operations take less than 10 ms with up to 20 attributes. On a Raspberry Pi, they are 10x slower (as expected), but still run at interactive speeds.

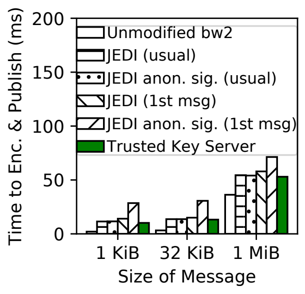
7.1.3 Performance of Immediate Revocation in JEDI

Fig. 6b shows the cost of JEDI's immediate revocation protocol (§5). A private key containing k leaves consists of $O(\log k + \log n)$ WKD-IBE secret keys where n is the total number of leaves. Therefore, the performance of immediate revocation depends primarily on the number of leaves.

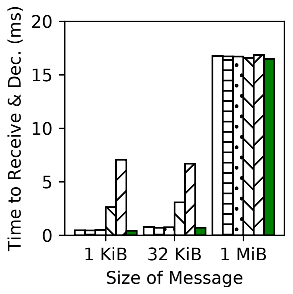
To encrypt a message, one WKD-IBE encryption is performed for each subtree needed to cover all unrevoked leaves. In general, encryption is $O(r \log \frac{n}{r})$, where r is the number of revoked leaves. Each key contains a set of *consecutive* leaves, so encryption is also $O(R \log \frac{n}{R})$, where R is the number of revoked JEDI keys. Decryption time remains almost the same, since only one WKD-IBE decryption is needed.

To benchmark revocation, we use a complete binary tree of depth 16 ($n = 65536$). The time to generate a new key for delegation is essentially independent of the number of leaves conveyed in that key, because $\log k \ll \log n$. We empirically confirmed this; the time to generate a key for delegation was constant at 2.4 ms on a laptop and 31 ms on a Raspberry Pi as the number of leaves in the key was varied from 5 to 1,000.

To benchmark encryption with revocation, we assume that there exist 2,048 users in the system each with 32 leaves. We measure encryption time with a pattern with 20 fixed slots (for URI and time) as we vary the number of revoked users. Fig. 6b shows that encryption becomes expensive when the revocation list is large (500 milliseconds on laptop and ≈ 5 seconds on Raspberry Pi). However, such an encryption only needs to be performed by a publisher when the URI, time, or revocation list changes; subsequent messages can reuse the underlying symmetric key (§5.5). Furthermore, the revocation list includes only revoked keys that match the (URI, time) pair being used, so it is not expected to grow very large.



(a) Encrypt/publish message



(b) Receive/decrypt message

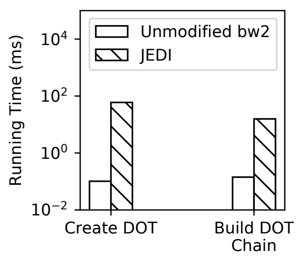
Figure 7: Critical-path operations in bw2, with/without JEDI

7.2 Performance of JEDI in bw2

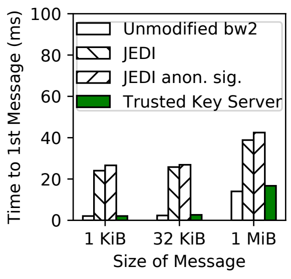
In bw2, the two critical-path operations are publishing a message to a URI, and receiving a message as part of a subscription. We measure the overhead of JEDI for these operations because they are core to bw2's functionality and would be used by any messaging application built on bw2. Our methodology is to perform each operation repeatedly in a loop, to measure the sustained performance (operations/second), and report the average time per operation (inverse). To minimize the effect of the network, the router was on the same link as the client, and the link capacity was 1 Gbit/s. In our experiments, we used a URI of length 6 and an Expiry tree of depth 6. We also include measurements from a strawman system with pre-shared AES keys—this represents the critical-path overhead of an approach based on the Trusted Key Server discussed in §2. Our results are in Fig. 7.

We implement the optimizations in §3.6.1, so only symmetric key encryption/decryption must be performed in the common case (labeled "usual" in the diagram). However, the symmetric keys will *not* be cached for the first message sent every hour, when the WKD-IBE pattern changes. A WKD-IBE operation must be performed in this case (labeled "1st message" in the diagram). For large messages, the cost of symmetric key encryption dominates. JEDI has a particularly small overhead for 1 MiB messages in Fig. 7b, perhaps because 1 MiB messages take several milliseconds to transmit over the network, allowing the client to decrypt a message while the router is sending the next message.

We also consider creating DOTs and initiating subscriptions, which are not in the critical path of bw2. These results are in Fig. 8 (note the log scale in Fig. 8a). Creating DOTs is slower with JEDI, because WKD-IBE keys are generated and included in the DOT. Initiating a subscription in bw2 requires forming a DOT chain; in JEDI, one must also derive a private key from the DOT chain. Fig. 8a shows the time to form a



(a) Create DOT, Build Chain



(b) Time to Query/Subscribe

Figure 8: Occasional bw2 operations, with and without JEDI

short one-hop DOT chain, and in the case of JEDI, includes the time to derive the private key. For JEDI's encryption (§3), these additional costs are incurred only by DOTs that grant permission to subscribe. With anonymous signatures, DOTs granting permission to publish incur this overhead as well, as WKD-IBE keys must be included. Fig. 8b puts this in context by measuring the end-to-end latency from initiating a subscription to receiving the first message (measured using bw2's "query" functionality).

For a DOT to be usable, it must be inserted into bw2's registry. This requires a blockchain transaction (not included in Fig. 8). An important consideration in this regard is *size*. In the unmodified bw2 system, a DOT that grants permission on `a/b/c/d/e/f` is 198 bytes. With JEDI, each DOT also contains multiple WKD-IBE keys, according to the time range. In the "worst case," where the start time of a DOT is Jan 01 at 01:00, and the end time is Dec 31 at 22:59, a total of 45 keys are needed. Each key is approximately 1 KiB (Table 6), so the size of this DOT is approximately 45 KiB.

Because bw2's registry of DOTs is implemented using blockchain smart contracts, the bandwidth for inserting DOTs is limited. Using JEDI would increase the size of DOTs as above, resulting in an approximately 100-400x decrease in aggregate bandwidth for creating DOTs. However, this can be mitigated by changing bw2 to not store DOTs directly in the blockchain. DOTs can be stored in untrusted storage, with only their hashes stored in the blockchain-based registry. Such a solution could be based on Swarm [84] or Filecoin [46].

7.3 Feasibility on Ultra Low-Power Devices

We use a commercially available sensor platform called "Hamilton" [4, 52] built around the Atmel SAMR21 system-on-chip (SoC). The SAMR21 costs approximately \$2.25 per unit [43] and integrates a low-power microcontroller and radio. The sensor platform we used in this study costs \$18 to manufacture [59]. For battery lifetime calculations, we assume that the platform is powered using a CR123A Lithium battery that provides 1400 mAh at 3.0 V (252 J of energy). Such a battery costs \$1. The SAMR21 is heavily constrained: it has only a 48 MHz CPU frequency based on the ARM Cortex-M0+ microarchitecture, and a total of only 32 KiB of data memory (RAM). Our goal is to validate that JEDI is practical for an ultra low-power sensor platform like Hamilton, in the context of a "sense-and-send" application in a smart building. Since most of the platform's cost (\$18) comes from the on-board transducers and assembly, rather than the SAMR21 SoC, *using an even more resource-constrained SoC would not significantly decrease the platform's cost.* An analogous argument applies to energy consumption, as the transducers account for more than half of Hamilton's idle current [59].

Table 2: CPU and power costs on the Hamilton platform

Operation	**Time**	**Average Current**
Sleep (Idle)	N/A	0.0063 mA
WKD-IBE Encrypt	6.50 s	10.2 mA
WKD-IBE Encrypt and Sign	9.89 s	10.2 mA

Hamilton/SAMR21 is on the lower end of platforms typically used for sense-and-send applications in buildings. Some older studies [44, 63] use even more constrained hardware like the TelosB; this is because those studies were constrained by hardware available at the time. Modern 32-bit SoCs, like the SAMR21, offer substantially better performance at a similar price/power point to those older platforms [59].

7.3.1 CPU Usage

Table 2 shows the time for encryption and anonymous signing in JEDI on Hamilton. The results use the optimizations discussed in §3.6 and §4.3, and include the time to "adjust" precomputed state. They indicate that symmetric keys can be encrypted and anonymously signed in less than 10 seconds. This is feasible given that encryption and anonymous signing occur rarely, once an hour, and need not be produced at interactive speeds in the normal "sense-and-send" use case.

7.3.2 Power Consumption

To calculate the impact on battery lifetime, we consider a "sense-and-send" application, in which the Hamilton device obtains readings from its sensors at regular intervals, and immediately sends the readings encrypted over the wireless network. We measured the average current consumed for varying sample intervals, when each message is encrypted with AES-CCM, without using JEDI ("AES Only" in Table 3). We estimate JEDI's average current based on the current, duration, and frequency (once per hour, for these estimates) of JEDI operations, and add it to the average current of the "AES Only" setup. Our estimates assume that the μTESLA-based technique in §4.3 is used to avoid attaching a digital signature to each message. We divide the battery's energy capacity by the result to compute lifetime. As shown in Table 3, JEDI decreases battery life by about 40-60%. Battery life is several years even with JEDI, acceptable for IoT sensor platforms.

JEDI's overhead depends primarily on the granularity of expiry times (one hour, for these estimates), *not* the sample interval. To improve power consumption, one could use a time tree with larger leaves, allowing principals to perform WKD-IBE encryptions and anonymous signatures less often. This would, of course, make expiry times coarser.

Table 3: Average current and expected battery life (for 1400 mAh battery) for sense-and-send, with varying sample interval

	AES Only	JEDI (enc)	JEDI (enc & sign)
10 s	32 µA / 5.1 y	50 µA / 3.2 y	60 µA / 2.6 y
20 s	20 µA / 8.1 y	38 µA / 4.2 y	48 µA / 3.3 y
30 s	15 µA / 10 y	34 µA / 4.7 y	44 µA / 3.6 y

7.3.3 Memory Budget

Performing WKD-IBE operations requires only 6.5 KiB of data memory, which fits comfortably within the 32 KiB of data memory (RAM) available on the SAMR21. The code space required for our implementation of WKD-IBE and BLS12-381 is about 74 KiB, which fits comfortably in the 256 KiB of code memory (ROM) provided by the SAMR21.

A related question is whether storing a hash chain in memory (as required for authenticated broadcast, §4.3) is practical. If we use a granularity of 1 minute for authenticated broadcast, the length of the hash chain is 60. At the start of an hour, one computes the entire chain, storing 10 hashes equally spaced along the chain, each separated by 5 hashes. As one progresses along the hash chain, one re-computes each set of 5 hashes one additional time. This requires storage for only 15 hashes ($<$ 4 KiB memory) and computation of only 105 hashes *per hour*, which is practical. One could possibly optimize performance further using *hierarchical hash chains* [53].

7.3.4 Impact of JEDI's Optimizations

JEDI's cryptographic optimizations (§3.6.2, §4.2.2, §4.3), which use WKD-IBE in a non-black-box manner, provide a 2-3x performance improvement. Our assembly optimizations (§6) provide an additional 4-5x improvement. Without both of these techniques, JEDI would not be practical on low-power sensors. Hybrid encryption and key reuse (§3.6.1), which let JEDI use WKD-IBE *rarely*, are also crucial.

7.4 Comparison to Other Systems

Table 4 compares JEDI to other systems and cryptographic approaches, particularly those geared toward IoT, in regard to security, expressivity and performance. We treat these existing systems as they would be used in a messaging system for smart buildings (§1). Table 4 contains quantitative comparisons to the cryptography used by these systems; for those schemes based on bilinear groups, we re-implemented them using our JEDI crypto library (§6.1) for a fair comparison.

Security. The owner of a resource is considered *trusted* for that resource, in the sense that an adversary who compromises a principal can read all of that principal's resources. In Table 4, we focus on whether a single component is trusted for *all* resources in the system. Note that, although Trusted Key Server (§2) and PICADOR [24] encrypt data in flight, granting or revoking access to a principal requires participation of an *online trusted party* to generate new keys.

Expressivity. PRE-based approaches, which associate public keys with users and support delegation via proxy re-encryption, are fundamentally coarse-grained—a re-encryption key allows *all* of a user's data to be re-encrypted. PICADOR [24] allows more fine-grained semantics, but does not enforce them cryptographically. ABE-based approaches typically do not support delegation beyond a single hop, whereas JEDI achieves multi-hop delegation. In ABE-based schemes, however, attributes/policies attached to keys can describe more complex sets of resources than JEDI. That said, a hierarchical resource representation is sufficient for JEDI's intended use case, namely smart cities; existing syndication systems for smart cities, which do not encrypt data and are unconstrained by the expressiveness of crypto schemes, choose a hierarchical rather than attribute-based representation (§1).

Performance. The Trusted Key Server (§2) is the most naïve approach, requiring an online trusted party to enforce all policy. Even so, JEDI's performance in the common case is the same as the Trusted Key Server (Fig. 7), because of JEDI's hybrid encryption—JEDI invokes WKD-IBE *rarely*. Even when JEDI invokes WKD-IBE, its performance is not significantly worse than PRE-based approaches. An alternative design for JEDI uses the GPSW KP-ABE construction instead of WKD-IBE, but it is significantly more expensive. Based Table 3, the power cost of a WKD-IBE operation *even when only invoked once per hour* contributes significantly to the overall energy consumption on the low-power IoT device; using KP-ABE instead of WKD-IBE would increase this power consumption by an order of magnitude, reducing battery life significantly.

In summary, existing systems fall into one of three categories. (1) The Trusted Key Server allows access to resources to be managed by arbitrary policies, but relies on a *central trusted party* who must be online whenever a user is granted access or is revoked. (2) PRE-based approaches, which permit sharing via re-encryption, cannot cryptographically enforce fine-grained policies or support multi-hop delegation. (3) ABE-based approaches, if carefully designed, *can* achieve the same expressivity as JEDI, but are substantially less performant and are not suitable for low-power embedded devices.

8 Related Work

We organize related work into the following categories.

Traditional Public-Key Encryption. SiRiUS [50] and Plutus [57] are encrypted filesystems based on traditional public-key cryptography, but they do not support delegable and qualifiable keys like JEDI. Akl et al. [2] and further work [36, 37] propose using key assignment schemes for access control in a hierarchy. A line of work [8, 9, 54, 85] builds on this idea to support both hierarchical structure and temporal access. Key assignment approaches, however, require the full hierarchy to be known at setup time, which is not flexible in the IoT setting. JEDI does not require this, allowing different subtrees of the hierarchy to be managed separately (§1.1, "Delegation").

Identity-Based Encryption. Tariq et al. [83] use Identity-Based Encryption (IBE) [19] to achieve end-to-end encryption in publish-subscribe systems, without the router's participation in the protocol. However, their approach does not

Table 4: Comparison of JEDI with other crypto-based IoT/cloud systems

Crypto Scheme / System	**Avoids Central Trust?**	**Expressivity**	**Performance**
Trusted Key Server (§2)	– No	+ Supports arbitrary policies (beyond hierarchies) – No delegation	+ $\approx$ 10 µs to encrypt 1 KiB message (same as JEDI in common case, faster for first message after key rotation) – Trusted party generates one key *per resource*
PRE (Lattice-Based), as used in PICADOR [24]	– No	+ Supports arbitrary policies (beyond hierarchies) – No delegation	+ $\approx$ 5 ms encrypt, $\approx$ 3 ms decrypt (similar to JEDI: 3-4 ms) – Trusted party must generate one key per sender-receiver pair
PRE (Pairing-Based), as used in Pilatus [80]	+ Yes	– Delegation is single-hop – Delegation is coarse (all-or-nothing) + Can compute aggregates on encrypted data	+ 0.6 ms encrypt, 1.3 ms re-encrypt, 0.5 ms decrypt (faster than JEDI: 3-4 ms) + Practical on constrained IoT device with crypto accelerator
CP-ABE [13]	+ Yes	+ Good fit for RBAC policies – Cannot support JEDI's hierarchy abstraction with delegation	+ Only symmetric crypto in common case – 14 ms encrypt for first time after key rotation (4-5x slower than JEDI: 3 ms)
KP-ABE, as used in Sieve [88]	+ Yes	+ Succinct delegation based on attributes – Delegation is single-hop	+ Only symmetric crypto in common case – 25 ms encrypt for first time after key rotation (8-9x slower than JEDI: 3 ms)
Delegable Large Univ. KP-ABE [51] (used in Alternative JEDI Design)	+ Yes	+ Generalizes beyond hierarchies and supports multi-hop delegation (subsumes JEDI)	+ Only symmetric crypto in common case – 60 ms encrypt for first time after key rotation (20x slower than JEDI: 3 ms) – Impractical for low-power sense-and-send
This paper: WKD-IBE [1] with Optimizations, as used in JEDI	+ Yes	+ Delegation is multi-hop + Succinct delegation of *subtrees* of resources (or more complex sets, §3.7) + Non-interactive expiry	+ After key rotation (e.g., once per hour), 3 ms encrypt, 4 ms decrypt (Fig. 6a) + Only symmetric crypto in common case + Practical for ultra low-power "sense-and-send" *without crypto accelerator*

support hierarchical resources. Further, encryption and private keys are on a credential-basis, so each message is encrypted multiple times according to the credentials of the recipients.

Wu et al. [92] use a prefix encryption scheme based on IBE for mutual authentication in IoT. Their prefix encryption scheme is different from JEDI, in that users with keys for identity `a/b/c` can decrypt messages encrypted with prefix identity `a`, `a/b` and `a/b/c`, but not identities like `a/b/c/d`.

Hierarchical Identity-Based Encryption. Since the original proposal of Hierarchical Identity-Based Encryption (HIBE) [49], there have been multiple HIBE constructions [17, 18, 48, 49] and variants of HIBE [1, 93]. Although seemingly a good match for resource hierarchies, HIBE cannot be used as a black box to efficiently instantiate JEDI. We considered alternative designs of JEDI based on existing variants of HIBE, but as we elaborate in Appendix C, each resulting design is either less expressive or significantly more expensive than JEDI.

Attribute-Based Encryption. A line of work [88, 95] uses Attribute-Based Encryption (ABE) [13,51] to delegate permission. For example, Yu et al. [95] and Sieve [88] use Key-Policy ABE (KP-ABE) [51] to control which principals have access to encrypted data in the cloud. Some of these approaches also provide a means to revoke users, leveraging proxy re-encryption to safely perform re-encryption in the cloud. Our work additionally supports hierarchically-organized resources and decentralized delegation of keys, which [95] and [88] do not address. As discussed in §7.4 and Appendix D, WKD-IBE is substantially more efficient than KP-ABE and provides enough functionality for JEDI. WKD-IBE could be a lightweight alternative to KP-ABE for some applications.

Other approaches prefer Ciphertext-Policy ABE (CP-ABE) [13]. Existing work [89, 90] combines HIBE with CP-ABE to produce Hierarchical ABE (HABE), a solution for sharing data on untrusted cloud servers. The "hierarchical" nature of HABE, however, corresponds to the hierarchical organization of domain managers in an enterprise, not a hierarchical organization of *resources* as in our work.

Proxy Re-Encryption. NuCypher KMS [42] allows a user to store data in the cloud encrypted under her public key, and share it with another user using Proxy Re-Encryption (PRE) [15]. While NuCypher assumes limited collusion among cloud servers and recipients (e.g., m of n secret sharing) to achieve properties such as expiry, JEDI enforces expiry via cryptography, and therefore remains secure against *any* amount of collusion. Furthermore, NuCypher's solution for resource hierarchies requires a keypair for each node in the hierarchy, meaning that the creation of resources is centralized. Finally, keys in NuCypher are not qualifiable. Given a key for `a/*`, one cannot generate a key for `a/b/*` to give to another principal.

PICADOR [24], a publish-subscribe system with end-to-end encryption, uses a lattice-based PRE scheme. However, PICADOR requires a central Policy Authority to specify access control, by creating a re-encryption key for every permitted pair of publisher and subscriber. In contrast, JEDI's access control is decentralized.

Revocation Schemes. Broadcast encryption (BE) [20–23, 40, 62, 72] is a mechanism to achieve revocation, by encrypting messages such that they are only decryptable by a specific set of users. However, these existing schemes do not support key qualification and delegation, and therefore, cannot be used in JEDI directly. Another line of work builds revocation directly into the underlying cryptography primitive, achieving Revocable IBE [16, 66, 77, 91], Revocable HIBE [67, 76, 78] and Revocable KP-ABE [10]. These papers use a notion of revocation in which URIs are revoked. In contrast, JEDI supports revocation at the level of keys. If multiple principals have access to a URI, and one of their keys is revoked, then the other principal can still use its key to access the resource. Some systems [12, 42] rely on the participation of servers or routers to achieve revocation.

Secure Reliable Multicast Protocol. Secure Reliable Multicast [68, 69] also uses a many-to-many communication model, and ensures correct data transfer in the presence of malicious routers. JEDI, as a protocol to *encrypt* messages, is complementary to those systems.

Authorization Services. JEDI is complementary to authorization services for IoT, such as bw2 [5], Vanadium [82], WAVE [6], and AoT [73], which focus on expressing authorization policies and enabling principals to prove they are authorized, rather than on encrypting data. Droplet [79] provides encryption for IoT, but does not support delegation beyond one hop and does not provide hierarchical resources.

An authorization service that provides secure in-band permission exchange, like WAVE [6], can be used for key distribution in JEDI. JEDI can craft keys with various permissions, while WAVE can distribute them without a centralized party by including them in its attestations.

9 Conclusion

In this paper, we presented JEDI, a protocol for end-to-end encryption for IoT. JEDI provides *many-to-many* encrypted communication on complex resource hierarchies, supports decentralized key delegation, and decouples senders from receivers. It provides expiry for access to resources, reconciles anonymity and authorization via anonymous signatures, and allows revocation via tree-based broadcast encryption. Its encryption and integrity solutions are capable of running on embedded devices with strict energy and resource constraints, making it suitable for the Internet of Things.

Availability

The JEDI cryptography library is available at https://github.com/ucbrise/jedi-pairing and our implementation of the JEDI protocol for bw2 is available at https://github.com/ucbrise/jedi-protocol.

Acknowledgments

We thank our anonymous reviewers and our shepherd William Enck for their invaluable feedback. We would also like to thank students from the RISE Security Group and BETS Research Group for giving us feedback on early drafts of this paper. This research was supported by Intel/NSF CPS-Security #1505773 and #20153754, DoE #DE-EE000768, California Energy Commission #EPC-15-057, 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] M. Abdalla, E. Kiltz, and G. Neven. Generalized key delegation for hierarchical identity-based encryption. Cryptology ePrint Archive, Report 2007/221.

[2] S. G. Akl and P. D. Taylor. Cryptographic solution to a problem of access control in a hierarchy. *TOCS*, 1983.

[3] M. P Andersen, G. Fierro, and D. E. Culler. System design for a synergistic, low power mote/BLE embedded platform. In *IPSN*, 2016.

[4] M. P Andersen, H.-S. Kim, and D. E. Culler. Hamilton - a cost-effective, low power networked sensor for indoor environment monitoring. In *BuildSys*, 2017.

[5] M. P Andersen, J. Kolb, K. Chen, D. E. Culler, and R. Katz. Democratizing authority in the built environment. In *BuildSys*, 2017.

[6] M. P Andersen, S. Kumar, M. AbdelBaky, G. Fierro, J. Kolb, H.-S. Kim, D. E. Culler, and R. A. Popa. WAVE: A decentralized authorization framework with transitive delegation. In *USENIX Security*, 2019.

[7] P. Arjunan, N. Batra, H. Choi, A. Singh, P. Singh, and M. B. Srivastava. SensorAct: A privacy and security aware federated middleware for building management. In *BuildSys*, 2012.

[8] M. J. Atallah, M. Blanton, N. Fazio, and K. B. Frikken. Dynamic and efficient key management for access hierarchies. In *TISSEC*, 2009.

[9] M. J. Atallah, M. Blanton, and K. B. Frikken. Incorporating temporal capabilities in existing key management schemes. In *ESORICS*, 2007.

[10] N. Attrapadung and H. Imai. Conjunctive broadcast and attribute-based encryption. In *ICPBC*, 2009.

[11] R. Barbulescu and S. Duquesne. Updating key size estimations for pairings. Cryptology ePrint Archive, Report 2017/334.

[12] S. Belguith, S. Cui, M. R. Asghar, and G. Russello. Secure publish and subscribe systems with efficient revocation. In *SAC*, 2018.

[13] J. Bethencourt, A. Sahai, and B. Waters. Ciphertext-policy attribute-based encryption. In *S&P*, 2007.

[14] A. Birgisson, J. G. Politz, Ú. Erlingsson, A. Taly, M. Vrable, and M. Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *NDSS*, 2014.

[15] M. Blaze, G. Bleumer, and M. Strauss. Divertible protocols and atomic proxy cryptography. In *EUROCRYPT*, 1998.

[16] A. Boldyreva, V. Goyal, and V. Kumar. Identity-based encryption with efficient revocation. In *CCS*, 2008.

[17] D. Boneh and X. Boyen. Efficient selective-ID secure identity-based encryption without random oracles. In *EUROCRYPT*, 2004.

[18] D. Boneh, X. Boyen, and E.-J. Goh. Hierarchical identity based encryption with constant size ciphertext. In *EUROCRYPT and Cryptology ePrint Archive*, 2005.

[19] D. Boneh and M. Franklin. Identity-based encryption from the Weil pairing. In *CRYPTO*, 2001.

[20] D. Boneh, C. Gentry, and B. Waters. Collusion resistant broadcast encryption with short ciphertexts and private keys. In *CRYPTO*, 2005.

[21] D. Boneh and B. Waters. A fully collusion resistant broadcast, trace, and revoke system. In *CCS*, 2006.

[22] D. Boneh, B. Waters, and M. Zhandry. Low overhead broadcast encryption from multilinear maps. In *CRYPTO*, 2014.

[23] D. Boneh and M. Zhandry. Multiparty key exchange, efficient traitor tracing, and more from indistinguishability obfuscation. *Algorithmica*, 2017.

[24] C. Borcea, A. B. D. Gupta, Y. Polyakov, K. Rohloff, and G. Ryan. PICADOR: End-to-end encrypted publish-subscribe information distribution with proxy re-encryption. *FGCS*, 2017.

[25] S. Bowe. BLS12-381: New zk-SNARK elliptic curve construction, 2018. https://z.cash/blog/new-snark-curve/.

[26] A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. P. Vasseur, and R. Alexander. RPL: IPv6 routing protocol for low-power and lossy networks. RFC, RFC Editor, 2012.

[27] D. Brunelli, I. Minakov, R. Passerone, and M. Rossi. POVOMON: An ad-hoc wireless sensor network for indoor environmental monitoring. In *EESMS*, 2014.

[28] bw2. https://github.com/immesys/bw2.

[29] B. Campbell. Introducing Hail, 2017. https://www.tockos.org/blog/2017/introducing-hail/.

[30] R. Canetti, S. Halevi, and J. Katz. A forward-secure public-key encryption scheme. In *EUROCRYPT*, 2003.

[31] R. Cheng, W. Scott, B. Parno, I. Zhang, A. Krishnamurthy, and T. Anderson. Talek: A private publish-subscribe protocol. Technical report, University of Washington CSE, 2016.

[32] J. H. Cheon. Security analysis of the strong Diffie-Hellman problem. In *EUROCRYPT*, 2006.

[33] Cisco. The Internet of things reference model. Technical report, Cisco, 2014.

[34] D. Clarke, J.-E. Elien, C. Ellison, M. Fredette, A. Morcos, and R. L. Rivest. Certificate chain discovery in SPKI/SDSI. *Journal of Computer Security*, 2001.

[35] H. Corrigan-Gibbs, D. Boneh, and D. Mazières. Riposte: An anonymous messaging system handling millions of users. In *S&P*, 2015.

[36] J. Crampton, N. Farley, G. Gutin, M. Jones, and B. Poettering. Cryptographic enforcement of information flow policies without public information. In *ACNS*, 2015.

[37] J. Crampton, K. Martin, and P. Wild. On key assignment for hierarchical access control. In *CSFW*, 2006.

[38] S. Dawson-Haggerty, X. Jiang, G. Tolle, J. Ortiz, and D. E. Culler. sMAP: A simple measurement and actuation profile for physical information. In *SenSys*, 2010.

[39] S. Dawson-Haggerty, A. Krioukov, J. Taneja, S. Karandikar, G. Fierro, N. Kitaev, and D. E. Culler. BOSS: Building operating system services. In *NSDI*, 2013.

[40] Y. Dodis and N. Fazio. Public key broadcast encryption for stateless receivers. In *DRM*, 2002.

[41] P. Dutta, D. E. Culler, and S. Shenker. Procrastination might lead to a longer and more useful life. In *HotNets*, 2007.

[42] M. Egorov and M. Wilkison. NuCypher KMS: Decentralized key management system. *CoRR*, 2017.

[43] DigiKey Electronics. ATSAMR21E18A-MU microchip technology. Feb. 8, 2019.

[44] M. C. Feldmeier. *Personalized Building Comfort Control*. PhD thesis, MIT, 2009.

[45] G. Fierro and D. E. Culler. XBOS: An extensible building operating system. Technical report, EECS Department, University of California, Berkeley, 2015.

[46] Filecoin. https://filecoin.io. Jan. 19, 2018.

[47] T. Frosch, C. Mainka, C. Bader, F. Bergsma, J. Schwenk, and T. Holz. How secure is TextSecure? In *EuroS&P*, 2016.

[48] C. Gentry and S. Halevi. Hierarchical identity based encryption with polynomially many levels. In *TCC*, 2009.

[49] C. Gentry and A. Silverberg. Hierarchical ID-based cryptography. In *ASIACRYPT*, 2002.

[50] E.-J. Goh, H. Shacham, N. Modadugu, and D. Boneh. SiRiUS: Securing remote untrusted storage. In *NDSS*, 2003.

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

[52] Hamilton IoT. https://hamiltoniot.com/.

[53] Y.-C. Hu, M. Jakobsson, and A. Perrig. Efficient constructions for one-way hash chains. In *ACNS*, 2005.

[54] H.-F. Huang and C.-C. Chang. A new cryptographic key assignment scheme with time-constraint access control in a hierarchy. *Computer Standards & Interfaces*, 2004.

[55] J. Hviid and M. B. Kjaergaard. Activity-tracking service for building operating systems. In *PerCom*, 2018.

[56] imix: Low-power IoT research platform, 2017. https://github.com/helena-project/imix.

[57] M. Kallahalla, E. Riedel, R. Swaminathan, Q. Wang, and K. Fu. Plutus: Scalable secure file sharing on untrusted storage. In *FAST*, 2003.

[58] Y. Kawahara, T. Kobayashi, M. Scott, and A. Kato. Barreto-Naehrig curves. Technical report, Internet-Draft draft-kasamatsu-bncurves-02. Internet Engineering Task Force., 2016. https://datatracker.ietf.org/doc/html/draft-kasamatsu-bncurves-02.

[59] H.-S. Kim, M. P Andersen, K. Chen, S. Kumar, W. J. Zhao, K. Ma, and D. E. Culler. System architecture directions for post-SoC/32-bit networked sensors. In *SenSys*, 2018.

[60] T. Kim and R. Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In *CRYPTO*, 2016.

[61] A. Krioukov, G. Fierro, N. Kitaev, and D. E. Culler. Building application stack (BAS). In *BuildSys*, 2012.

[62] A. Lewko, A. Sahai, and B. Waters. Revocation systems with very small private keys. In *S&P*, 2010.

[63] C. Li, Z. Li, M. Li, F. Meggers, A. Schlueter, and H. B. Lim. Energy efficient HVAC system with distributed sensing and control. In *ICDCS*, 2014.

[64] B. Libert, T. Peters, and M. Yung. Group signatures with almost-for-free revocation. In *CRYPTO*, 2012.

[65] B. Libert, T. Peters, and M. Yung. Scalable group signatures with revocation. In *EUROCRYPT*, 2012.

[66] B. Libert and D. Vergnaud. Adaptive-ID secure revocable identity-based encryption. In *CT-RSA*, 2009.

[67] W. Liu, J. Liu, Q. Wu, B. Qin, D. Naccache, and H. Ferradi. Compact CCA2-secure hierarchical identity-based broadcast encryption for fuzzy-entity data sharing. Cryptology ePrint Archive, Report 2016/634.

[68] D. Malkhi, M. Merritt, and O. Rodeh. Secure reliable multicast protocols in a WAN. *Dist. Computing*, 2000.

[69] D. Malkhi and M. Reiter. A high-throughput secure reliable multicast protocol. *Computer Security*, 1997.

[70] A. Mehanovic, T. H. Rasmussen, and M. B. Kjærgaard. Brume - a horizontally scalable and fault tolerant building operating system. In *IoTDI*, 2018.

[71] R. C. Merkle. A certified digital signature. In *ASIACRYPT*, 1989.

[72] D. Naor, M. Naor, and J. Lotspiech. Revocation and tracing schemes for stateless receivers. In *CRYPTO*, 2001.

[73] A. L. M. Neto, A. L. F. Souza, I. Cunha, M. Nogueira, I. O. Nunes, L. Cotta, N. Gentille, A. A. F. Loureiro, D. F. Aranha, H. K. Patil, and L. B. Oliveira. AoT: Authentication and access control for the entire IoT device life-cycle. In *SenSys*, 2016.

[74] Particle Mesh. https://www.particle.io/mesh. Feb. 2, 2019.

[75] A. Perrig, R. Szewczyk, V. Wen, D. E. Culler, and J. D. Tygar. SPINS: Security protocols for sensor networks. In *MobiCom*, 2001.

[76] J. H. Seo and K. Emura. Efficient delegation of key generation and revocation functionalities in identity-based encryption. In *CT-RSA*, 2013.

[77] J. H Seo and K. Emura. Revocable identity-based encryption revisited: Security model and construction. In *PKC*, 2013.

[78] J. H. Seo and K. Emura. Revocable hierarchical identity-based encryption: History-free update, security against insiders, and short ciphertexts. In *CT-RSA*, 2015.

[79] H. Shafagh, L. Burkhalter, S. Duquennoy, A. Hithnawi, and S. Ratnasamy. Droplet: Decentralized authorization for IoT data streams. *CoRR*, 2018.

[80] H. Shafagh, A. Hithnawi, L. Burkhalter, P. Fischli, and S. Duquennoy. Secure sharing of partially homomorphic encrypted IoT data. In *SenSys*, 2017.

[81] Solace cloud. https://solace.com. Jan. 17, 2018.

[82] A. Taly and A. Shankar. Distributed authorization in Vanadium. In *FOSAD VIII*, 2016.

[83] M. A. Tariq, B. Koldehofe, and K. Rothermel. Securing broker-less publish/subscribe systems using identity-based encryption. *TPDS*, 2014.

[84] V. Tron, A. Fischer, and N. Johnson. Smash-proof: Auditable storage for Swarm secured by masked audit secret hash. Technical report, Ethersphere, 2016.

[85] W.-G. Tzeng. A time-bound cryptographic key assignment scheme for access control in a hierarchy. *TKDE*, 2002.

[86] J. van den Hooff, D. Lazar, M. Zaharia, and N. Zeldovich. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *SOSP*, 2015.

[87] VOLTTRON. https://volttron.org/. Jan. 23, 2019.

[88] F. Wang, J. Mickens, N. Zeldovich, and V. Vaikuntanathan. Sieve: Cryptographically enforced access control for user data in untrusted clouds. *NSDI*, 2016.

[89] G. Wang, Q. Liu, and J. Wu. Hierarchical attribute-based encryption for fine-grained access control in cloud storage services. In *CCS*, 2010.

[90] G. Wang, Q. Liu, J. Wu, and M. Guo. Hierarchical attribute-based encryption and scalable user revocation for sharing data in cloud servers. *Computers & Security*, 2011.

[91] Y. Watanabe, K. Emura, and J. H. Seo. New revocable IBE in prime-order groups: Adaptively secure, decryption key exposure resistant, and with short public parameters. In *CT-RSA*, 2017.

[92] D. J. Wu, A. Taly, A. Shankar, and D. Boneh. Privacy, discovery, and authentication for the Internet of things. In *ESORICS*, 2016.

[93] D. Yao, N. Fazio, Y. Dodis, and A. Lysyanskaya. ID-based encryption for complex hierarchies with applications to forward security and broadcast encryption. In *CCS*, 2004.

[94] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *INFOCOM*, 2002.

[95] S. Yu, C. Wang, K. Ren, and W. Lou. Achieving secure, scalable, and fine-grained data access control in cloud computing. In *INFOCOM*, 2010.

[96] T. Zachariah, N. Klugman, B. Campbell, J. Adkins, N. Jackson, and P. Dutta. The Internet of things has a gateway problem. In *HotMobile*, 2015.

[97] Zigbee gateway. https://www.zigbee.org/zigbee-for-developers/zigbee-gateway/. Feb. 13, 2019.

A JEDI's Optimizations to WKD-IBE

The purpose of Appendix A is twofold. First, it largely reproduces the construction of WKD-IBE provided in §3.2 of [1], which we used in JEDI, for readers who would like additional context. Second, it fully explains the "Precomputation with Adjustment" optimizations (§3.6.2) that JEDI uses for fast encryption and signatures on low-power platforms.

A.1 Construction with our Optimizations

We present the WKD-IBE construction in §3.2 of [1]. For completeness, we include the extension to signatures described in §4.2, as well as our observation in §3.1 that the encryption algorithm in [1] can be extended to work with arbitrary patterns.

The construction is based on bilinear groups. $\mathbb{G}$ and $\mathbb{G}_T$ are cyclic groups of prime order p, and they are related by a bilinear map $e : \mathbb{G} \times \mathbb{G} \to \mathbb{G}_T$. The security parameter κ is related to the number of bits of p. We implemented WKD-IBE using BLS12-381, an *asymmetric* bilinear group whose bilinear map is of the form $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$. We denote $\mathbb{G}_1$ the smaller and faster of the two source groups (elliptic curve over $\mathbb{F}_p$). The construction of WKD-IBE was originally defined for symmetric bilinear groups. Our description of the construction below shows how we mapped the construction onto an asymmetric bilinear group.

Setup(1^ℓ): Select $g \xleftarrow{\$} \mathbb{G}_2$ and $g_2, g_3, h_1, \dots, h_\ell, h_s \xleftarrow{\$} \mathbb{G}_1$. Then select $\alpha \xleftarrow{\$} \mathbb{Z}_p$ and let $g_1 = g^\alpha$. Output: $\mathsf{Params} = (g, g_1, g_2, g_3, h_1, \dots, h_\ell, h_s)$ and $\mathsf{MasterKey} = g_2^\alpha$.

KeyDer(K, S): If K is the master key, take $K = g_2^\alpha$. Select $r \xleftarrow{\$} \mathbb{Z}_p$. The private key for the pattern S is the following triple:

$$\left(g_2^\alpha \cdot \left(g_3 \cdot \prod_{(i,a_i) \in \mathrm{fixed}(S)} h_i^{a_i} \right)^r, \quad g^r, \quad \left\{ (j, h_j^r) \right\}_{j \in \mathrm{free}(S)} \right).$$

If K is not the master key, then parse K as (k_0, k_1, B), where $B = \{(i, b_i)\}$. Select $t \xleftarrow{\$} \mathbb{Z}_p$. The private key for S is:

$$\left(k_0 \cdot \left(g_3 \cdot \prod_{(i,a_i) \in \mathrm{fixed}(S)} h_i^{a_i} \right)^t \cdot \prod_{\substack{(i,a_i) \in \mathrm{fixed}(S) \\ (i,b_i) \in B}} b_i^{a_i}, \quad g^t \cdot k_1, \quad \left\{ (j, h_j^t \cdot b_j) \right\}_{j \in \mathrm{free}(S)} \right).$$

Observe that the resulting key is identically distributed, regardless of whether or not the input key K is the master key.

Encrypt(S, m): Here, $m \in \mathbb{G}_T$. Select $s \xleftarrow{\$} \mathbb{Z}_p$ and output

$$\left(e(g_1, g_2)^s \cdot m, \quad g^s, \quad \left(g_3 \cdot \prod_{(i,a_i) \in \mathrm{fixed}(S)} h_i^{a_i} \right)^s \right).$$

Decrypt(K, C): Parse the key K as (k_0, k_1, B), and the ciphertext C as (X, Y, Z). Output

$$X \cdot e(k_1, Z) \cdot e(Y, k_0)^{-1}.$$

To support encryption over arbitrary patterns (§3.1), we only compute the product over fixed slots, just as is done in the BBG HIBE construction [18]. In contrast, the original WKD-IBE construction requires all slots in the pattern to be fixed, and iterates over all slots. The proof technique from [18], namely padding the selected ID with zeros, can be used here to modify the proof of WKD-IBE [1] to account for our optimization that allows free slots to be used in encryption.

A.2 Construction of WKD-IBE Signatures

We originally explained anonymous signatures in §4.2.1 by (1) dedicating one of the slots in a pattern for signing messages, and (2) defining signature generation as a call to **KeyDer** that fills that dedicated slot. We then proposed improvements to the signature scheme, to make it constant size (§4.2.2) and make the verification procedure more efficient (§4.3). We formally describe our optimizations to the signature algorithm below. Note that we added an extra term to the public parameter h_s that represents the slot dedicated to signing messages, and an analogous element (s, b_s) to the third component of each secret key. It was not present in the original WKD-IBE construction, and is not used for encryption in Appendix A.1.

Sign(K, m): Parse the key K as (k_0, k_1, B), where $(s, b_s) \in B$. Let S be the pattern corresponding to K. Select $t \xleftarrow{\$} \mathbb{Z}_p$ and output

$$\left(k_0 \cdot \left(g_3 \cdot h_s^m \cdot \prod_{(i,a_i)\in\text{fixed}(S)} h_i^{a_i}\right)^t \cdot b_s^m, \quad g^t \cdot k_1\right)$$

Verify(S, σ, m): Parse the signature σ as (s_0, s_1). Check:

$$e(s_0, g) \stackrel{?}{=} e(g_1, g_2) \cdot e\left(g_3 \cdot h_s^m \cdot \prod_{(i,a_i)\in\text{fixed}(S)} h_i^{a_i}, \quad s_1\right)$$

In contrast to optimized procedures above, the naïve signature algorithm has **Sign**(K,m) = **KeyDer**(K,T), and **Verify**(S,σ,m) = (**Decrypt**$(\sigma,$ **Encrypt**$(T,m^*)) \stackrel{?}{=} m^*$) for $m^* \xleftarrow{\$} \mathbb{Z}_p^*$, where T is the same as S except that $T(s) = m$, the message being signed. The modification we make is that (1) the signature contains only the first two components of **KeyDer**(K,T) (since the third component is not used for decryption), and (2) the verification procedure checks that σ is a private key corresponding to T more efficiently than encrypting and decrypting a random message.

Finally, note that the **Sign** function can be generalized to allow a key with pattern P to produce a signature for pattern S if P matches S. This can be done trivially by first applying **KeyDer** to obtain a key for S, and calling the **Sign** on the existing key. Our implementation supports this **GeneralizedSign** functionality more efficiently, as follows:

GeneralizedSign(K,S,m): Parse the key K as (k_0,k_1,B), where $(s,b_s) \in B$. Select $t \xleftarrow{\$} \mathbb{Z}_p$ and output

$$\left(k_0 \cdot \left(g_3 \cdot h_s^m \cdot \prod_{(i,a_i)\in\text{fixed}(S)} h_i^{a_i}\right)^t \cdot b_s^m \cdot \prod_{\substack{(i,a_i)\in\text{fixed}(S)\\(i,b_i)\in B}} b_i^{a_i}, \quad g^t \cdot k_1\right)$$

A.3 Precomputation with Adjustment

We formally explain the precomputation optimization introduced in §3.6.2 and §4.3.

A.3.1 Precomputation with Adjustment for Encryption

We define the new WKD-IBE operations as follows (as before, Params is an implicit parameter):

Precompute(S): Output

$$g_3 \cdot \prod_{(i,a_i)\in\text{fixed}(S)} h_i^{a_i}$$

AdjustPrecomputed(Q_S, S, T): Q_S is the existing precomputed value, S is the pattern it corresponds to, and T is the pattern whose precomputed value to compute. Output

$$Q_S \cdot \prod_{\substack{(i,b_i)\in\text{fixed}(T)\\ i\in\text{free}(S)}} h_i^{b_i} \cdot \prod_{\substack{(i,a_i)\in\text{fixed}(S)\\ i\in\text{free}(T)}} h_i^{-a_i} \cdot \prod_{\substack{(i,a_i)\in\text{fixed}(S)\\(i,b_i)\in\text{fixed}(T)\\ a_i\neq b_i}} h_i^{b_i-a_i}$$

EncryptPrepared(Q_S, m): Here, $m \in \mathbb{G}_T$. Select $s \xleftarrow{\$} \mathbb{Z}_p$ and output

$$(e(g_1,g_2)^s \cdot m, \quad g^s, \quad Q_S^s).$$

The above routines are used as described in §3.6.2. Observe that the output of **Encrypt**(S,m) and the output of **EncryptPrepared**(**Precompute**$(S)$$,m)$ are distributed identically—security of this optimization relies on this fact.

A.3.2 Precomputation with Adjustment for Signatures

As explained in §4.3, we think of **KeyDer** in two parts, **NonDelegableKeyDer** and **ResampleKey**:

NonDelegableKeyDer(K,S): Parse K as (k_0,k_1,B), where $B = \{(i,b_i)\}$. Output:

$$\left(k_0 \cdot \prod_{\substack{(i,a_i)\in\text{fixed}(S)\\(i,b_i)\in B}} b_i^{a_i}, \quad k_1, \quad \{(j,b_j)\}_{j\in\text{free}(S)}\right).$$

ResampleKey(K,S): Parse K as (k_0,k_1,B). Sample $t \xleftarrow{\$} \mathbb{Z}_p$ and output

$$\left(k_0 \cdot \left(g_3 \cdot \prod_{(i,a_i)\in\text{fixed}(S)} h_i^{a_i}\right)^t, g^t \cdot k_1, \{(j, h_j^t \cdot b_j)\}_{j\in\text{free}(S)}\right).$$

NonDelegableKeyDer and **ResampleKey** are the "two parts" of **KeyDer** in the sense that $\{$**KeyDer**$(K,S)\}$ = $\{$**ResampleKey**(**NonDelegableKeyDer**$(K,S),S)\}$ where the distributions are over the sampled randomness.

We can take advantage of these functions to accelerate signing of messages. Note the similarity between **Sign** and **ResampleKey**. The setup we consider is that a principal has a key for some pattern R representing a URI prefix and time prefix. It will repeatedly sign messages with a pattern S representing at a fully-qualified URI and specific time, where R matches S. The next signature will be on pattern T which shares the same URI as S but corresponds to the next leaf in the time tree. The naïve algorithm is to call **QualifyKey** to obtain a key for S and then call **Sign**. The key idea behind the optimization is to instead call **NonDelegableKeyDer** to obtain a pseudo-key for S (which is not safe to delegate), and then create a signature for that. Observe that the resulting signature is distributed in exactly the same way whether the naïve or optimized method is used. (§4.3, which explains **Sign** as simply being a call to **KeyGen**, explains this technique as splitting **KeyDer** into two parts, **NonDelegableKeyDer** and **ResampleKey**.)

Now that we have described the signature process in terms of two calls, one to **NonDelegableKeyDer** and another to **Sign**, we describe how to apply Precomputation with Adjustment to each of these operations. **Sign** can be accelerated using the same precomputed value we used to accelerate encryption. We have already shown how to "adjust" this precomputed value from S to T.

SignPrepared(K, Q_S, m): Parse the key K as (k_0, k_1, B). Let S be the pattern corresponding to K; Q_S must be the precomputed value corresponding to S. Select $t \xleftarrow{\$} \mathbb{Z}_p^*$ and output:

$$\left(k_0 \cdot (h_s^m \cdot Q_S)^t, \quad g^t \cdot k_1\right)$$

Finally, we explain how the result of **NonDelegableKeyDer** can be adjusted from pattern S to pattern T. The procedure also requires the parent key (whose pattern we denote R), on which **NonDelegableKeyDer** was called to obtain the key corresponding to pattern S.

AdjustNonDelegable(P, C, S, T): Parse the parent key P as P as (p_0, p_1, B) where $B = \{(i, b_i)\}$. Parse the child key C as $C = (k_0, k_1, Z)$. S is the pattern corresponding to C, and T is the pattern that the resulting key will correspond to. Output:

$$\left(k_0 \cdot \prod_{\substack{(i,t_i)\in\text{fixed}(T)\\ i\in\text{free}(S)}} b_i^{t_i} \cdot \prod_{\substack{(i,s_i)\in\text{fixed}(S)\\ i\in\text{free}(T)}} b_i^{-s_i} \cdot \prod_{\substack{(i,s_i)\in\text{fixed}(S)\\ (i,t_i)\in\text{fixed}(T)}} b_i^{t_i - s_i}, \quad k_1, \quad \{(j, b_j)\}_{j\in\text{free}(T)}\right).$$

To sign a message each hour, JEDI maintains the result of **Precompute**, Q_S (as it does for encryption), and also the result of **NonDelegableKeyGen**, C, derived from its key. Then it adjusts both values, using **AdjustPrecomputed** and **AdjustNonDelegable**, when the pattern used to sign changes. To sign a message m, it computes **SignPrepared**(C, Q_S, m).

As an additional optimization, we only compute the first two elements of the output of **NonDelegableKeyDer** and **AdjustNonDelegableKeyDer** when using it to produce signatures.

B Revocation in JEDI using the SD Method

The SD algorithm for tree-based broadcast encryption is introduced in prior work [40, 72]. In this section, we explain how to extend SD algorithm to support delegation. We follow the same approach to apply this extension in JEDI as we did for the CS method in §5.

B.1 Overview of SD Method

We first provide a brief overview of the SD method, as described in [40]. Both the CS and SD method use a technique called *Subset Cover*. Let N be the (finite) set of all users. A family of subsets $\mathcal{S}$ is defined over N. Each each element S_i of $\mathcal{S}$ is a subset of N, and corresponds to a keypair $(\mathsf{PK}_{S_i}, \mathsf{SK}_{S_i})$. Each user is given the secret key SK_{S_i} for every set S_i containing that user. Now, suppose that someone wants to encrypt a message such that it is visible only to a subset of users, denoted B. To achieve this, she must find a *subset cover* for B: sets $B_1, \ldots, B_p \in \mathcal{S}$ such that $B = \bigcup_{j=1}^{p} B_i$. Then, she encrypts her message under the public key PK_{B_j} for each B_j in the subset cover. Only users in B have the secret key for one of the B_j to decrypt the message.

The CS method (as described in §5.3) uses a subset family $\mathcal{S}$, where each subset is a complete subtree of the binary tree over the users. The SD method uses a different subset family $\mathcal{S}$ than the CS method, in which each leaf belongs to more subsets. Each subset $S_{ij} \in \mathcal{S}$ is defined in terms of two nodes, v_i and v_j, where v_j is a descendant of v_i. S_{ij} contains all leaves in the subtree rooted at v_i but not in the subtree rooted at v_j. Each leaf now belongs to $\sum_{k=1}^{\log(n)} (2^k - k)$ different subsets, and therefore has $O(n)$ secret keys.

Each subset S_{ij} is associated with an ID in HIBE as follows. The first component of the ID is $\mathsf{ID}(v_i)$. The remaining components of the ID are $\mathsf{ID}(v_j)$, where each bit occupies one component, which makes it possible to generate the private key for S_{ik} from the private key for S_{ij}, as long as v_k is a descendant of v_j.

If a leaf belongs to both S_{ij} and S_{ik}, it only needs to be given S_{ij}. With this optimization, each leaf only has $O(\log^2 n)$ HIBE secret keys.

For encryption, the sender must find a subset cover over all unrevoked leaves, as in the CS method. The algorithm for finding the optimal subset cover is introduced in [72]. [72] also proves that only $O(r)$ subsets (and therefore $O(r)$ encryptions) are needed.

B.2 Extension of SD Method for Delegation

We follow the same idea in §5.3 that each key corresponds to a range of consecutive leaves, and the delegation and revocation follow the same rule. In contrast to CS method, we use WKD-IBE instead of HIBE to reduce the storage requirement.

B.2.1 Analysis of Private Key Storage

A principal with a set of consecutive leaves denoted LF must be able to generate private keys for all subsets $S_{jk} \in \mathcal{S}$ where $S_{jk} \cap \mathsf{LF} \neq \varnothing$.

We define S_j as the set of leaves in the subtree rooted at v_j, and $\mathsf{copath}(v)$ as in a Merkle Tree [71]. Then define $\mathsf{CoPath}(\mathsf{LF})$ as follows:

$$\mathsf{CoPath}(\mathsf{LF}) = \mathsf{copath}(\mathsf{leaf}(\mathsf{lf}_{\min})) \cup \mathsf{copath}(\mathsf{leaf}(\mathsf{lf}_{\max}))$$

where $\mathsf{lf}_{\min}$ stands for the first leaf in LF and $\mathsf{lf}_{\max}$ stands for the last leaf in LF. $\mathsf{LeftChild}(v_i)$ and $\mathsf{RightChild}(v_i)$ stands for the left child node and the right child node of v_i respectively.

Then a principal with LF only needs to store the private keys for subsets S_{jk} satisfying the following properties:

- $S_{jk} \cap \mathsf{LF} \neq \varnothing$ and
  1. If $S_{\mathsf{LeftChild}(v_j)} \cap \mathsf{LF} \neq \varnothing$ and $S_{\mathsf{RightChild}(v_j)} \cap \mathsf{LF} \neq \varnothing$, then v_k must satisfy $v_k = \mathsf{LeftChild}(v_j)$ or $v_k = \mathsf{RightChild}(v_j)$.
  2. If $S_{\mathsf{LeftChild}(v_j)} \cap \mathsf{LF} = \varnothing$ or $S_{\mathsf{RightChild}(v_j)} \cap \mathsf{LF} = \varnothing$, then v_k must satisfy $v_k \in \mathsf{CoPath}(\mathsf{LF})$.

There will be $O(k)$ subsets S_{jk} satisfying the first condition, and $O(\log^2 n)$ subsets S_{jk} satisfying the second condition. Thus the total number of keys will be $O(k + \log^2 n)$. Note that we leverage HIBE as in the original SD Method to achieve $O(\log^2 n)$ keys for the second kind of subset.

However, we cannot use HIBE to reduce storage for the first kind of subsets because for two subsets S_{ab} and S_{cd} in the first type, either $v_a \neq v_c$ or v_b is the sibling of v_d (no hierarchical relations). Recall that HIBE can only help reduce storage when $v_a = v_c$ and v_b is the ancestor or descendant of v_d. However, we observe that v_a might be the ancestor or descendant of v_c, which means we can use WKD-IBE to support another concurrent hierarchy. We discuss this in Appendix B.2.2.

B.2.2 Optimization

We will represent the $O(k)$ subsets satisfying the first condition using only $O(\log k)$ keys. We will do this by introducing a hierarchy for the first index (the i in S_{ij}). However, we are already using a hierarchy for the second index (the j in S_{ij}), and HIBE cannot support two hierarchies simultaneously. Therefore, we will use WKD-IBE to combine the two hierarchies, as we did in §3.2.

We use a WKD-IBE system with $2\log(n)+1$ slots. We use the first $\log(n)+1$ slots to support a hierarchy for the first index, and the remaining $\log n$ slots to support a hierarchy for the second index. Each subset S_{ij} is associated with a pattern in WKD-IBE as follows. The first hierarchy encodes $\mathsf{ID}(v_i)$ (each bit in a separate slot), followed by `$`, a termination symbol. The second hierarchy encodes $\mathsf{ID}(v_j)$, where each bit is stored in a separate slot.

With this construction, we preserve all of the functionality from before. The second hierarchy representing the second index can be extended, just as HIBE in original SD method. As before, the first index cannot be modified, because the terminator symbol `$` prevents the first hierarchy from being extended. Alternatively, limited delegation can be used for this purpose (as in §5.3), to avoid using one component of the WKD-IBE pattern for the terminator symbol `$`. Note that there are two logical hierarchies in this construction, and only one of them needs to be made unqualifiable via limited delegation.

Now, we can more efficiently represent the secret keys for the $O(k)$ subsets satisfying the first condition above. Observe that the k consecutive leaves can be grouped into $O(\log k)$ consecutive subtrees. Let $\mathsf{TR}(\mathsf{LF})$ be the set containing the root nodes of these subtrees. For each node $v_i \in \mathsf{TR}(\mathsf{LF})$, we must provide the private key corresponding to $S_{i\mathsf{LeftChild}(v_i)}$ and $S_{i\mathsf{RightChild}(v_i)}$, without including the terminator symbol `$` in the attribute set. From these $O(\log k)$ WKD-IBE keys, a principal can obtain all $O(k)$ keys corresponding to subsets S_{ij} satisfying the first condition above. This reduces the total number of secret keys for k consecutive leaves from $O(k + \log^2 n)$ to $O(\log k + \log^2 n)$.

B.2.3 Using Delegable SD Method in JEDI

As in §5.3.3, we use the first $\ell_1 + \ell_2$ slots for URI and expiry, and use the remaining slots to instantiate the above revocation protocol. Different from the delegable CS method (§5.3.2), the number of slots used for revocation is $\ell_3 = 2\log(n) + 1$, and they are used by the delegable SD method as two independent hierarchies. So the total number of WKD-IBE keys is $O((\log k + \log^2 n) \cdot \log T)$, where T is the length of the time range for expiry.

C Alternative Designs of JEDI using HIBE

We explore alternative designs we could have used for JEDI, focusing on existing cryptographic primitives we considered using instead of WKD-IBE.

C.1 Hierarchical Identity-Based Encryption

Given that JEDI represents URIs and time as hierarchies, Hierarchical Identity-Based Encryption (HIBE) [18] may seem like a natural building block to use. We can encode a URI in an ID for HIBE, just as we did for WKD-IBE. For example, the URI prefix `a/b/*` can be encoded into an ID as `("a", "b")`. This preserves the crucial property that the private key for a URI prefix can be used to generate the private key for any URI with that prefix. The same thing works for expiry: for example, the timestamp June 08, 2017 at 6 AM could be encoded into an ID as `("2017", "June", "08", "06")`.

However, HIBE cannot *simultaneously* support a URI hierarchy and an expiry hierarchy. A simple approach would be to concatenate the IDs. For example, the key for the URI prefix `a/b/*` and time prefix `2018/Jun/*` would have the ID `("2018", "Jun", "a", "b")`. However, this idea is flawed: only the URI can be extended, not the expiry time. The same problem applies to the URI, if we put the URI before the time in the ID. Another possible approach is to interleave the resource hierarchy and time hierarchy, using metadata to distinguish the elements. In this setup, each (resource, time) pair corresponds to multiple IDs in the HIBE system—all possible interleavings of the URI the Expiry IDs. However, for a URI of length m and a time of length n, there are exponentially many IDs, $\frac{(m+n)!}{m! \cdot n!}$, and each message sent with that URI and time must be encrypted under all of those IDs. Therefore, this approach is infeasible.

Another strawman is to use two HIBE systems, one for URIs and one for expiry. Each message is encrypted twice, using the URI ID in the first system and again using the time ID in the second system. During delegation, each principal is provided with a key from the first system for the URI, and a set of keys from the second system for the time range.

The problem is that this approach is not collusion-resistant: a principal who is given two delegations, one for the correct URI that has expired, and one for the wrong URI that has not expired, can decrypt messages by combining keys from different delegations.

C.2 Variants of HIBE other than WKD-IBE

Existing work [93] has proposed extending HIBE to MHIBE, which supports ID-based encryption for multiple concurrent hierarchies. We could use MHIBE in JEDI to combine the URI hierarchy with the expiry hierarchy. However, the proposed MHIBE schemes are significantly less performant than WKD-IBE: for two hierarchies, they have quadratically-sized private keys and ciphertexts. Size and performance degrade exponentially in the number of hierarchies. Furthermore, a formal treatment of MHIBE is not provided.

Another extension is forward secure HIBE [18, 93], or fs-HIBE for short. The BBG construction of fs-HIBE [18] has linear size and performance. We considered using its mechanism for forward security *in reverse*, to achieve expiry with HIBE. However, the fs-HIBE construction has linear-size ciphertexts and linear-time decryption, whereas WKD-IBE has constant-size ciphertexts and constant-time decryption. In the context of a real system, this is important: **Encrypt** and **Decrypt** are used in the critical path, so encryption time, decryption time, and ciphertext size must be as small as possible. In contrast, **Delegate** is only used occasionally, so the size of private keys is less important.

Most importantly, WKD-IBE is a more powerful primitive than either MHIBE or fs-HIBE. In particular, WKD-IBE supports the + wildcard for URIs and timestamps (Appendix F.1), which MHIBE and fs-HIBE do not.

D Building Block Comparison: KP-ABE

In Key-Policy Attribute-Based Encryption (KP-ABE), a message is encrypted with a set of attributes. An attribute set is like a string of bits; each attribute is either present in the set (1) or not present (0). Private keys are generated with an *access tree*, which can be thought of as a circuit. A private key can decrypt a message if its access tree, evaluated on the bits representing the attribute set of the message, evaluates to 1.

We are interested in KP-ABE with two properties:

1. **Delegable.** Given the private key for an access tree, one can generate a private key for a more restrictive access key, and delegate it to another principal.
2. **Large Universe.** The space of attributes $\mathcal{A}$ is exponentially large in the security parameter κ. This is similar to Identity-Based Encryption (IBE) [19], as any string of bytes can be hashed to an attribute.

The GPSW construction [51] of KP-ABE, based on bilinear groups, satisfies these properties. In fact, KP-ABE with these two properties subsumes WKD-IBE. A pattern T in WKD-IBE can be converted to an attribute set in Delegable Large Universe KP-ABE by hashing each non-$\perp$ component of T, concatentated with its index, to an attribute in KP-ABE. Private keys in WKD-IBE can be expressed as an access tree consisting of a single many-input AND gate.[3]

Table 5: Performance comparison of HIBE, WKD-IBE, and KP-ABE in terms of pairings and exponentiations. We omit operations that can be precomputed once for all IDs (attribute sets) in the HIBE/WKD-IBE/KP-ABE system. **KeyDer^1** indicates deriving the new key from the master key, and **KeyDer^2** indicates the other case.

Operation	Scheme	Pairings	Exponentiations
Encrypt	HIBE	0	$3+r$
Encrypt	WKD-IBE	0	$3+r$
Encrypt	KP-ABE	0	$2+r\cdot(\ell+3)$
Decrypt	HIBE	2	$\leq r$
Decrypt	WKD-IBE	2	$\leq r$
Decrypt	KP-ABE	$r+1$	$2r$
KeyDer^1	HIBE	0	$\ell+2$
KeyDer^1	WKD-IBE	0	$\ell+2$
KeyDer^1	KP-ABE	0	$r\cdot(\ell+5)$
KeyDer^2	HIBE	0	$(r-n)+\ell+2$
KeyDer^2	WKD-IBE	0	$(r-n)+\ell+2$
KeyDer^2	KP-ABE	0	$2n+r\cdot(\ell+5)$

The subsections below compare BBG HIBE, WKD-IBE, and GPSW KP-ABE. Although we include HIBE for the sake of comparison, note that HIBE is not expressive enough to realize the JEDI protocol (as explained in Appendix C).

D.1 Performance Comparison

We compare the performance of KP-ABE, WKD-IBE, and HIBE in terms of the number of exponentiations and pairings, the most expensive operations in the elliptic curves. This is shown in Table 5. ℓ is the total number of attributes that can be used for a single message (the implicit argument to **Setup**). For Encrypt, Decrypt and KeyDer^1, r is the number of attributes of the key or ciphertext. For KeyDer^2, n is the number of attributes of the starting key, and r is the number of attributes of the ending key. This shows that WKD-IBE's performance is theoretically better than KP-ABE's performance. Furthermore, WKD-IBE is just efficient as HIBE, even though WKD-IBE is more expressive than HIBE. As discussed in Appendix C, HIBE is not expressive enough to efficiently instantiate JEDI.

D.2 Size Comparison

We list the size of ciphertexts and private keys in Table 6: r is the number of attributes in the ciphertext or private key, and ℓ is the maximum number of slots or attributes used to encrypt a message. Note that ciphertexts in WKD-IBE are constant size, whereas ciphertexts in KP-ABE are linear.

[3]Ciphertext-Policy ABE (CP-ABE) is not suitable for this construction. This is because attributes cannot be added to secret keys during delegation, as per the security guarantees of CP-ABE.

Table 6: Size comparison of HIBE, WKD-IBE, and KP-ABE in terms of number of group elements. For elliptic curves that we used, elements of $\mathbb{G}_1$ are 48 B each, elements of $\mathbb{G}_2$ are 96 B each, and elements of $\mathbb{G}_T$ are 576 B each.

Object	Scheme	$\mathbb{G}_1$	$\mathbb{G}_2$	$\mathbb{G}_T$
Ciphertext	HIBE	1	1	1
Ciphertext	WKD-IBE	1	1	1
Ciphertext	KP-ABE	r	1	1
Private Key	HIBE	$\ell - r + 1$	1	0
Private Key	WKD-IBE	$\ell - r + 1$	1	0
Private Key	KP-ABE	r	r	0

E Formal Definitions and Proofs

In this section, we present our formal definitions of JEDI's security guarantees and the corresponding proofs. Our proofs use the notion of IND-sWKID-CPA security defined in §3 of [1]. They also depend on a property of the construction of WKD-IBE called *history-independence*, which we explain below.

E.1 Definition of History-Independence and Proof of Theorem 2

Informally, a WKD-IBE construction is *history-independent* if, for any fixed pattern S, the result of **KeyDer** to produce a key with pattern S, assuming that the starting key is either the master key or corresponds to a pattern that matches S, is distributed in exactly the same way regardless of the particular starting key used. The idea is that, given a key for a specified pattern S, one learns nothing about the sequence of **KeyDer** operations that produced the key.

We formally define history-independence below:

Definition 2 (History-Independence)**.** *A WKD-IBE construction is said to be history-independent if, for every pattern S, and for any two well-formed keys k_1 (corresponding to pattern P_1) and k_2 (corresponding to pattern P_2) in the same WKD-IBE system such that P_1 matches S and P_2 matches S, it holds that*

$$\{\textbf{KeyDer}(k_1, S)\} = \{\textbf{KeyDer}(k_2, S)\}$$

where the distributions are over the randomness sampled internally by **KeyDer**. *If k_1 (respectively, k_2) is the master key, the pattern P_1 (respectively, P_2) is one where all slots are free.*

Below, we show that the construction of WKD-IBE presented in Appendix A, which JEDI uses, satisfies this property.

Theorem 3. *The construction of WKD-IBE presented in Appendix A is history-independent.*

Proof of Theorem 3. We will show that for any pattern S and any well-formed key k corresponding to a pattern P that matches S, it holds that

$$\{\textbf{KeyDer}(k, S)\} = \left\{ \left(g_2^{\alpha} \cdot \left(g_3 \cdot \prod_{(i,a_i) \in \text{fixed}(S)} h_i^{a_i} \right)^r, \; g^r, \; \left\{ (j, h_j^r) \right\}_{j \in \text{free}(S)} \right) \right\}_{r \xleftarrow{\$} \mathbb{Z}_p}$$

Because the formula on the right-hand side of the above equation only depends on S and the public parameters (not the particular key k), this is sufficient to demonstrate history-independence of the WKD-IBE construction.

We handle the proof in two cases:

Case 1. Suppose that k is the master key. Then the above result is true by definition, according to the formula given for **KeyDer** in Appendix A.1.

Case 2. Suppose that k is not the master key. Then, because k is well-formed, we can write that

$$k = \left(g_2^{\alpha} \cdot \left(g_3 \cdot \prod_{(i,a_i) \in \text{fixed}(P)} h_i^{a_i} \right)^{r_0}, \; g^{r_0}, \; \left\{ (j, h_j^{r_0}) \right\}_{j \in \text{free}(P)} \right)$$

for some fixed $r_0 \in \mathbb{Z}_p$. By applying the formula for **KeyDer** in Appendix A.1, we can see that the key output by **KeyDer** has the form

$$\left(g_2^{\alpha} \cdot \left(g_3 \cdot \prod_{(i,a_i) \in \text{fixed}(S)} h_i^{a_i} \right)^{r_0+t}, \; g^{r_0+t}, \; \left\{ (j, h_j^{r_0+t}) \right\}_{j \in \text{free}(S)} \right)$$

for $t \xleftarrow{\$} \mathbb{Z}_p$. Because $r_0 + t$ is uniformly distributed in $\mathbb{Z}_p$, the output key has the desired distribution (take $r = r_0 + t$). □

Theorem 2 follows directly from the fact that the WKD-IBE construction used in JEDI is history-independent: each "signature" in WKD-IBE is the same as a private key generated with a special slot filled in with the message being signed. Therefore, signatures inherit the history-independence of keys, resulting in the property in Theorem 2. With the proposed improvement to make signatures constant size (§4.2.2), the signature consists of just the first two terms of the resulting private key, but it remains history-independent nonetheless.

For completeness, we prove Theorem 2 below, using the same notation for signatures established in Appendix A.1. The proof is very similar to the proof of Theorem 3

Proof of Theorem 2. We will show that for any pattern S, key k corresponding to pattern S, and message m, it holds that

$$\{\textbf{Sign}(k, m)\} = \left\{ \left(g_2^{\alpha} \cdot \left(g_3 \cdot h_s^m \cdot \prod_{(i,a_i) \in \text{fixed}(S)} h_i^{a_i} \right)^r, \; g^r \right) \right\}_{r \xleftarrow{\$} \mathbb{Z}_p}$$

Because the right-hand side of the above equation depends only on S and the public parameters (not the particular key k), this is sufficient to prove Theorem 2 (that any two keys corresponding to S produce the same signature distribution).

Observe that for a well-formed key k,

$$k = \left(g_2^{\alpha} \cdot \left(g_3 \cdot \prod_{(i,a_i) \in \text{fixed}(S)} h_i^{a_i} \right)^{r_0}, g^{r_0}, \left\{ (j, h_j^{r_0}) \right\}_{j \in \text{free}(S)} \right)$$

for some fixed $r_0 \in \mathbb{Z}_p$. Applying the formula for **Sign** in Appendix A.1, the signature has the form

$$\left(g_2^{\alpha} \cdot \left(g_3 \cdot h_s^m \cdot \prod_{(i,a_i) \in \text{fixed}(S)} h_i^{a_i} \right)^{r_0+t}, g^{r_0+t} \right)$$

for $t \xleftarrow{\$} \mathbb{Z}_p$. Because $r_0 + t$ is uniformly distributed in $\mathbb{Z}_p$, the output signature has the desired distribution (take $r = r_0 + t$). □

Finally, we note that Theorem 2 and its proof easily extend to the "generalized" signatures (**GeneralizedSign**) discussed in Appendix A.1, where a key for pattern P can generate a signature for *another* pattern S where P matches S. In this case, the theorem guarantees that for any message m, pattern S, and two well-formed keys k_1 and k_2 with patterns P_1 and P_2 (in the same resource hierarchy), **if** P_1 matches S and P_2 matches S, **then** signatures for pattern S and message m generated using k_1 are distributed identically to signatures for pattern S and message m generated using k_2, even if $P_1 \neq P_2$. The proof for this generalization of Theorem 2 is very similar to the proofs of Theorem 3 and Theorem 2.

E.2 Proof of Theorem 1

Below, we prove Theorem 1 from §3.8. Some intuition behind the proof is that the challenger in the game in Theorem 1 (which is also the adversary in the IND-sWKID-CPA game [1]), maintains, for each principal, the set of keys it has. It *lazily* requests these keys from the IND-sWKID-CPA challenger as principals are compromised. Therefore, it maintains (1) a *key set* for each principal, storing the keys requested from the IND-sWKID-CPA challenger, and (2) a *pattern set* for each principal, storing patterns corresponding to additional keys that the principal would have in the normal course of JEDI, but which have not been requested from the IND-sWKID-CPA challenger yet. Requesting keys lazily is crucial because an uncompromised principal in Theorem 1 may possess a secret key capable of decrypting the challenge ciphertext.

Proof of Theorem 1. We show that, given an adversary $\mathcal{A}$ with non-negligible advantage in the game in Theorem 1, one can construct an algorithm $\mathcal{B}$ with non-negligible advantage in the IND-sWKID-CPA security game [1]. We denote as $\mathcal{C}$ the IND-sWKID-CPA security challenger. $\mathcal{B}$ maintains the following state: (2) a mapping from principal name (in $\{0,1\}^*$) to a key set for that principal, and (3) a mapping from principal name (in $\{0,1\}^*$) to a pattern set for that principal. These two maps are initialized as follows; each has a single entry for the name corresponding to the authority. The authority's key set is empty, and its pattern set contains one element, namely a pattern containing $\perp$ in all components (i.e., with all slots free).

$\mathcal{B}$ first runs the game with $\mathcal{A}$ as the challenger. $\mathcal{A}$ specifies the pair (URI, time) that it will attack at the beginning of the game. $\mathcal{B}$ parses (URI, time) into the pattern S^* and gives it to $\mathcal{C}$. $\mathcal{C}$ generates the master key pair $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{Setup}$ and gives B the master public key mpk. $\mathcal{B}$ forwards mpk to $\mathcal{A}$. For any of three queries from $\mathcal{A}$ in Phase 1, $\mathcal{B}$ processes it as following:

- $\mathcal{A}$ asks $\mathcal{B}$ to create a principal: $\mathcal{B}$ returns a fresh name in $\{0,1\}^*$ corresponding to the new principal. $\mathcal{B}$ creates mappings from this name to an empty set, for both the key set and pattern set, indicating that this new principal has not been delegated any keys.
- $\mathcal{A}$ asks $\mathcal{B}$ for the key set of a principal p: $\mathcal{B}$ finds in its local state the key set and pattern set for p. For each pattern in p's pattern set, it queries $\mathcal{A}$ for the corresponding WKD-IBE secret key. It adds each WKD-IBE secret key to p's key set, and then replaces p's pattern set in its local state with an empty set. Then it returns the keys in p's key set to $\mathcal{A}$. Note that $\mathcal{B}$ will not query $\mathcal{C}$ the secret key for a pattern that matches S^*, because, in the game in Theorem 1, $\mathcal{A}$ is not allowed to request a key set containing a key whose URI and time match the challenge pair (URI, time). Also note that the keys given to $\mathcal{A}$ are distributed exactly as they would be in the JEDI protocol, because the underlying WKD-IBE scheme is assumed to be history-independent.
- $\mathcal{A}$ asks an principal p to make a delegation of $\mathcal{A}$'s choice of another principal q: $\mathcal{B}$ finds in its local state the key set and pattern set for p. $\mathcal{B}$ obtains the pattern corresponding to each key in p's key set. Let M be the set containing those patterns. $\mathcal{B}$ computes the set N, which is the union of M and p's pattern set. Based on the patterns in N, $\mathcal{B}$ computes the patterns corresponding to the keys that p would generate and delegate to q. For each such key, $\mathcal{B}$ adds the corresponding pattern to q's pattern set.

At the end of Phase 1, $\mathcal{A}$ outputs two equal-length challenge messages m_0 and m_1, and sends them to $\mathcal{B}$. $\mathcal{B}$ then forwards m_0 and m_1 to $\mathcal{C}$. $\mathcal{C}$ chooses a random bit b, and sends $\mathcal{B}$ the ciphertext of m_b. $\mathcal{B}$ then forwards the ciphertext to $\mathcal{A}$.

In Phase 2, $\mathcal{A}$ makes additional queries as in Phase 1, and $\mathcal{C}$ can process them as before.

Finally, $\mathcal{A}$ will return the bit b'. $\mathcal{B}$ returns b' to $\mathcal{C}$. Because every response that $\mathcal{B}$ makes to $\mathcal{A}$ is distributed identically to the results of actually playing the game in Theorem 1, $\mathcal{A}$ will guess $b' = b$ with non-negligible advantage. Thus, $\mathcal{B}$ wins the IND-sWKID-CPA game with non-negligible advantage. □

E.3 Security Guarantee of JEDI's Revocation

Theorem 4. *Suppose revocation in JEDI is instantiated with a Selective-ID CPA-secure [1, 17, 30], history-independent*

WKD-IBE scheme. Then, there exists no probabilisitc polynomial-time adversary $\mathcal{A}$ who can win the following security game against a challenger $\mathcal{C}$ with more than negligible advantage:

Initialization. *$\mathcal{A}$ selects a (URI, time) pair and a list $\mathcal{L}$ of revoked leaves to attack.*

Setup. *$\mathcal{C}$ gives $\mathcal{A}$ the public parameters of the JEDI instance.*

Phase 1. *$\mathcal{A}$ can make three types of queries to $\mathcal{C}$.*

1. $\mathcal{A}$ asks $\mathcal{C}$ to create a principal; $\mathcal{C}$ returns a name in $\{0,1\}^*$, which $\mathcal{A}$ can use to refer to that principal in future queries. A special name exists for the authority.

2. $\mathcal{A}$ can ask $\mathcal{C}$ for the key set of any principal; $\mathcal{C}$ gives $\mathcal{A}$ the keys that the principal has. The only restriction is that, at the time this query is made, every key in the requested key set whose URI and time are prefixes of the challenge (URI, time) must have leaves that are entirely in the challenge list $\mathcal{L}$.

3. $\mathcal{A}$ can ask $\mathcal{C}$ to make any principal to make a delegation of $\mathcal{A}$'s choice to another principal. The new principal may have restricted pattern or fewer leaves.

Challenge. *When $\mathcal{A}$ chooses to end Phase 1, it sends $\mathcal{C}$ two messages, m_0 and m_1, of the same length. Then $\mathcal{C}$ chooses a random bit $b \in \{0,1\}$, encrypts m_b under the challenge (URI, time) pair and list $\mathcal{L}$ of revoked leaves, and gives $\mathcal{A}$ all of the ciphertexts.*

Phase 2. *$\mathcal{A}$ can make additional queries as in Phase 1.*

Guess. *$\mathcal{A}$ outputs $b' \in \{0,1\}$, and wins the game if $b = b'$. The advantage of an adversary $\mathcal{A}$ is $\left|\Pr[\mathcal{A} \text{ wins}] - \frac{1}{2}\right|$.*

Proof. The proof of Theorem 4 is somewhat trickier than the proof of Theorem 1 because encrypted messages in JEDI, when revocation is used, consist of multiple WKD-IBE ciphertexts. Therefore, we use a hybrid argument. The hybrid game $\mathcal{H}^{(0)}$ is one that the adversary $\mathcal{A}$ has no chance of winning. Hybrid $\mathcal{H}^{(n)}$ is a game that perfectly simulates WKD-IBE's revocation protocol. We prove using security of the underlying WKD-IBE scheme that for all $i \in \{1,...,n\}$, the difference between $\mathcal{A}$'s advantage in hybrid game $\mathcal{H}^{(i-1)}$ and hybrid game $\mathcal{H}^{(i)}$ is negligible.

The number n, which controls the number of hybrids, is defined to be the size of the subset cover, which depends on the list $\mathcal{L}$ that $\mathcal{A}$ declares in the Initialization phase. Note that this is the same as the number of WKD-IBE ciphertexts in an encrypted message with revocation list $\mathcal{L}$.

We define the hybrid game $\mathcal{H}^{(i)}$, for $i \in \{0,...,n\}$, as identical to the game given in Theorem 4, with the following difference. The JEDI ciphertext returned to $\mathcal{A}$ in the Challenge phase consists of n WKD-IBE ciphertexts; $\mathcal{H}^{(i)}$ generates the first i ciphertexts correctly, and then replaces each of the remaining $n-i$ ciphertexts with an encryption of 0 under the same pattern. Observe that $\mathcal{A}$ has no chance of winning $\mathcal{H}^{(0)}$, because the challenge ciphertext is chosen independently of the bit b chosen by the challenger. Also observe that $\mathcal{H}^{(n)}$ is identical to the game described in Theorem 4.

All that remains to prove is that the difference between $\mathcal{A}$'s advantage in game $\mathcal{H}^{(i-1)}$ and game $\mathcal{H}^{(i)}$ is negligible for all $i \in \{1,...,n\}$. We do so via a reduction: given a probabilistic polynomial-time adversary $\mathcal{A}$ whose difference in advantage is non-negligible, we construct a probabilistic polynomial-time adversary $\mathcal{B}$ that wins the IND-sWKID-CPA game with non-negligible probability. In our reduction, $\mathcal{B}$ acts as the challenger in the hybrid game; we denote the challenger in the IND-sWKID-CPA game as $\mathcal{C}$.

In the Initialization phase, $\mathcal{A}$ specifies the pair (URI, time) and revocation list $\mathcal{L}$ that it will attack. Then, $\mathcal{B}$ computes the subset cover over all leaves not in $\mathcal{L}$, and selects ID_i, the ID of ith subset in the subset cover. $\mathcal{B}$ parses (URI, time) and ID_i into the pattern S^* and gives it to $\mathcal{C}$. $\mathcal{C}$ generates the master key pair $(\mathsf{mpk}, \mathsf{msk}) \leftarrow \mathsf{Setup}$ and gives B the master public key mpk. $\mathcal{B}$ forwards mpk to $\mathcal{A}$.

For any of three queries from $\mathcal{A}$ in Phase 1, $\mathcal{B}$ processes it as it does in the Proof of Theorem 1. The only difference is that $\mathcal{A}$ also specifies which leaves are included in each delegation (query #3), and $\mathcal{B}$ takes this into account when generating the pattern set N. Note that, because of the "subset cover" property used by tree-based broadcast encryption, the WKD-IBE patterns used to query $\mathcal{A}$ for WKD-IBE keys will never match S^*.

At the end of Phase 1, $\mathcal{A}$ outputs two equal-length challenge messages m_0 and m_1, and sends them to $\mathcal{B}$. $\mathcal{B}$ then chooses a random bit b^*. $\mathcal{B}$ computes the JEDI ciphertext, which consists of n WKD-IBE ciphertexts as follows. To compute the jth WKD-IBE ciphertext, where $1 \le j \le i-1$, it encrypts 0 with the pattern corresponding to the challenge (URI, time) and ID_j. To compute jth WKD-IBE ciphertext, where $i+1 \le j \le n$, it encrypts m_{b^*} with the pattern corresponding to the challenge (URI, time) and ID_j. To compute the ith WKD-IBE ciphertext, it forwards 0 and m_{b^*} to $\mathcal{C}$. $\mathcal{C}$ chooses a random bit b, and sends $\mathcal{B}$ either an encryption of 0 or an encryption of m_{b^*} depending on b.the ciphertext of m_b. $\mathcal{B}$ uses this as the ith ciphertext. It assembles the n WKD-IBE ciphertexts, computed as above, into a JEDI ciphertext, and forwards them to $\mathcal{A}$. Note that if $b = 0$, then $\mathcal{B}$ played game $\mathcal{H}^{(i-1)}$, and if $b = 1$, then $\mathcal{B}$ played game $\mathcal{H}^{(i)}$.

In Phase 2, $\mathcal{A}$ makes additional queries as in Phase 1, and $\mathcal{C}$ can process them as before.

Finally, $\mathcal{A}$ will return the bit b'. $\mathcal{B}$ checks if $b' = b^*$. If $b' = b^* *$, then $\mathcal{B}$ guesses that $b = 1$; otherwise, it guesses that $b = 0$. It sends its guess to $\mathcal{C}$. Because $\mathcal{A}$ is assumed to have a non-negligible difference in advantage between $\mathcal{H}^{(i-1)}$ and $\mathcal{H}^{(i)}$, $\mathcal{B}$'s advantage in the IND-sWKID-CPA game is non-negligible. □

E.4 Adaptive Security

A natural question is how to achieve adaptive security. As has been observed for IBE [17], HIBE [18], and WKD-IBE [1], hashing each component of the ID results in adaptive security, but with a loss of security exponential in the size of the hash. However, if the hash function is modeled as a random oracle, and the number of slots in WKD-IBE is logarithmic in the

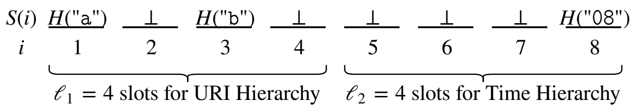

S(i) H("a") ⊥ H("b") ⊥ ⊥ ⊥ ⊥ H("08")
i 1 2 3 4 5 6 7 8
ℓ1 = 4 slots for URI Hierarchy
ℓ2 = 4 slots for Time Hierarchy

Figure 9: Pattern S for a private key granting access to `a/+/b/*` at 8 AM every day. The figure uses 8 slots for space reasons; JEDI is meant to be used with more slots (e.g., 20).

security parameter, then the loss in security is polynomial [1] (assuming that the number of slots ℓ is logarithmic in the security parameter). Given that we use a hash function to map URI/time to a pattern (§3.4), this analysis applies to JEDI.

F Extensions

We present two extensions to JEDI's core encryption protocol (§3): (1) generalized subtrees with wildcards in the middle of a URI, and (2) forward secrecy.

F.1 Beyond Simple Hierarchies

Thus far, we have considered only the `*` wildcard at the end of a URI. With WKD-IBE, we can also place a `+` wildcard in the middle of a URI, allowing a single component of the URI to remain unspecified. For example, the URI `a/+/b` matches all URIs of length 3 where the first component is `a` and the third component is `b`; the second component could be anything. To implement the `+` wildcard, we fill in the components corresponding to `+` with $\perp$.

The `+` wildcard is useful in real applications. For example, in the "smart buildings" setting, one could imagine a resource hierarchy of the form `buildingA/floor2/room/sensor_id/reading_type`, where `reading_type` could be either `temp` or `hum`. The `+` wildcard allows one to delegate permission to see only the temperature readings in a building, by granting permission on the URI `buildingA/+/+/+/temp`. It is also useful for the time hierarchy. An organization may want to give an employee access to a resource from 8 AM to 5 PM every day, which can be accomplished by using the `+` wildcard for the slots corresponding to the year, month, and day. See Fig. 9.

F.2 Forward Secrecy

Forward secrecy is the property that if a subscriber's decryption key is compromised, the attacker should only be able to decrypt *new* messages visible to the subscriber, not old messages sent before the key was compromised.

Forward secrecy using HIBE has been studied in [30]. We can apply the same idea to our construction via a straightforward extension to our mechanism for expiry. In our construction of expiry, each subscriber has a collection of keys for each URI or URI prefix it can access that give it access over a time range $[t_1, t_2]$, which can be qualified to any smaller time range $[t_3, t_4]$ where $t_1 \leq t_3 \leq t_4 \leq t_2$. To achieve forward security, each subscriber qualifies the keys for each URI, at each unit of time, to only be valid starting at the *current time* until the same expiry time, and then discards the old keys. This guarantees that, if a key is stolen, it cannot be used to decrypt messages published before the current time.