Democratizing Authority in the Built Environment

Michael P Andersen, John Kolb, Kaifei Chen, David E. Culler, Randy Katz
University of California, Berkeley
{m.andersen,jkolb,kaifei,culler,randy}@cs.berkeley.edu

ABSTRACT

Operating systems and applications in the built environment have relied upon central authorization and management mechanisms which restrict their scalability, especially with respect to administrative overhead. We propose a new set of primitives encompassing syndication, security, and service execution that unifies the management of applications and services across the built environment, while enabling participants to individually delegate privilege across multiple administrative domains with no loss of security or manageability. We show how to leverage a decentralized authorization syndication platform to extend the design of building operating systems beyond the single administrative domain of a building. The authorization system leveraged is based on blockchain smart contracts to permit decentralized and democratized delegation of authorization without central trust. Upon this, a publish/subscribe syndication tier and a containerized service execution environment are constructed. Combined, these mechanisms solve problems of delegation, federation, device protection and service execution that arise throughout the built environment. We leverage a high-fidelity city-scale emulation to verify the scalability of the authorization tier, and briefly describe a prototypical democratized operating system for the built environment using this foundation.

CCS CONCEPTS

• **Security and privacy** → **Access control**; **Authorization**; *Authentication*; • **Computer systems organization** → *Embedded and cyber-physical systems*;

KEYWORDS

Built Environment, Syndication, Microservices, Federation

ACM Reference format:
Michael P Andersen, John Kolb, Kaifei Chen, David E. Culler, Randy Katz. 2017. Democratizing Authority in the Built Environment. In *Proceedings of The 4th International Conference on Systems for Energy-Efficient Built Environments, Delft, The Netherlands, November 8-9, 2017 (BuildSys),* 10 pages.
https://doi.org/10.1145/3137133.3137151

BuildSys, November 8-9, 2017, Delft, The Netherlands

ACM ISBN 978-1-4503-5544-5/17/11…$15.00
https://doi.org/10.1145/3137133.3137151

1 INTRODUCTION

Over the past several years, building operating systems (BOS) have been developed to provide an execution environment for applications that operate safely on the physical plant of a building, transforming it into a rich cyberphysical system (e.g., BAS [21], sMAP [13], BuildingDepot [1], BOSS [14], Mortar.io [22], BEMOSS [23], Niagara [20], VOLTTRON [2]).

While this work has made great strides in supporting energy efficiency, human comfort, and grid integration of buildings, three themes emerge that are both outstanding shortcomings of BOS and critical to extending from buildings (or campuses of buildings) to the broader built environment – (i) natural delegation of authority and its enforcement, (ii) federation, and (iii) protection. Furthermore, each of these are present not just for individuals, but for persistent computational processes and devices operating on their behalf.

Delegation: Within the building context, organizationally, the campus facility manager delegates certain authority to building managers, each of whom may delegate (logically) the control of certain regions of a building to its tenant organizations, who may further delegate to individual office occupants. However, little of this delegation is actually supported by building management systems. Often the individual does not even have a way to set desired temperature; she instead makes a request up the chain of authority. The building manager may be authorized on a central server to adjust zone setpoints and schedules, but she must issue requests to facilities managers to make deeper adjustments like supply air temperature. In a more modern BOS, the occupant may be able to open a webpage or app to adjust the temperature schedule for their particular office, but authorization to do so is based on verifying the identity of the individual (or possibly the identity of a device, e.g., a touchpad, in the room) used in accordance with an access control list (ACL). If the occupant loans the office to a visitor, there may be no way to also 'loan' the authority to adjust its temperature or lighting schedule other than getting the visitor inserted and later removed from the directory. The delegation problem is amplified as we consider the lifecycle of the building, operations performed on common spaces, temporary authorizations for events, etc. These issues are further amplified across the built environment. The individual is delegated different access in their apartment, work setting, gym, and public space, for example. Furthermore, as we move to intelligent environments, the individual may want to delegate a portion of her authority in each of these places to her computational agents, so out-of-band human communication is not sufficient.

Federation: Even within a building, its subsystems (HVAC, lighting, electrical monitoring, security) have typically been disconnected, managed by different control systems. Most BOS integrate these and many will integrate the resulting systems over multiple buildings in a common administrative domain, say a campus or a property management company. The challenge is integration across distinct administrative domains. This arises in settings as simple as

demand response, where the issuer of the critical event (e.g., the independent system operator) is distinct from the building owner, its occupants, the utility, or the energy services aggregator. As electric vehicle charging is incorporated, the set of stakeholders increases, as with the incorporation of municipal and industrial processes, especially in conjunction with fluctuating renewable supplies. The mechanisms to allow agents in one administrative domain to take particular limited actions in another, say adjusting a temperature or lighting set-point, is quite ad hoc. We cannot expect all of these to refer their authorizations to a common central authority, nor can we expect to deal with many disparate authorizing entities. Certainly, the vendors of each of the different devices cannot be the authorizing entity (as we see today with Nest, Amazon Echo, Google Home). Nor can we expect all the parties to be on-line and participating whenever a delegation is performed.

Protection: As more of the built environment is connected and intelligent, the danger arises that more attack surfaces and threats are presented. Embedded devices tend to be particularly sensitive to DOS attacks because of limited processing power, often limited connection bandwidth, and largely unattended operation. Several examples in the literature [12, 24] show current vulnerabilities to these threats and modern BOS have not fundamentally changed the picture. They typically apply modern network security, sit behind firewalls, avoid open ports, and may use VPNs to cross LANs.

Execution containers: Lying beneath these issues is the simple fact that the building blocks of cyberphysical systems are numerous persistent computational processes – drivers, gateways, controllers, and so on. Each of these resides in some execution container and needs to be managed (initiated, monitored, controlled, restarted, etc.), as does the container itself. In building a foundation for delegation, protection, and federation in higher level services these principles can be utilized internally in the support for these services.

To address these issues in the expansion from intelligent buildings to intelligence throughout the built environment, we explore incorporating delegation, protection, and federation into the communication substrate of the syndication layer. The idea is that in any attempt to publish to a resource, the entity making the attempt must present a proof of authorization (over multiple levels of delegation, across administrative domains) that can be easily verified by the router forming the syndication layer and by the recipient subscribers. Entities can delegate rights to access resources to other entities without engaging any central authority, in fact without any communication with any entities whatsoever, i.e., no assumption that the parties involved are on-line. We implement these using the Wave system (contributed by a separate paper) which enforces them cryptographically via blockchain smart contracts.

The design is rooted in three concepts: entities, namespaces, and delegation-of-trust (DoT). In Wave, an entity is simply a key pair, identified by its (public) verifying key, that may represent any participant: individuals, devices, services, applications, components of the system implementation, and so on. A namespace is a hierarchy of resources that is identified as and owned by its authorizing entity, which has full access to all resources within it. Each resource is identified by a path rooted in the namespace identity, i.e., **namespace/path...**. Rights include the ability to publish to a resource or subscribe to a collection of resources (described by a subtree expression) in a namespace. Thus, each resource is logically a stream of messages. It may also provide a persistent message, i.e., state. In general, a sensor device publishes to the resource that represents it; an actuator subscribes to a resource expression representing its interface. An entity may delegate a permission to a resource in a namespace to another entity. Doing so places a delegation-of-trust edge from granter to grantee in the *global permission graph*. Such action does not involve any interaction with the namespace, the resource, or the entities involved. It may not even be valid at the time it is created (which in practice is essential). When an entity publishes (subscribes) to a resource (resource tree) it must present a proof of authorization consisting of a valid DoT path in the permission graph from the authorizing entity of the namespace to itself encompassing the resource (tree). These concepts are realized in a set of microcontracts on an Ethereum blockchain [28], an Agent daemon that abstracts this from applications and a Router daemon that performs overlay routing and syndication.

2 BUILDING OPERATING SYSTEMS BACKGROUND

Where traditional Building Management Systems (BMS) in commercial buildings provide a central point of supervisory control that is connected to and providing set-points for many dedicated direct controllers and provide a limited set of services (status screens, schedules, logging, trending, alarms), Building Operating Systems (BOS) seek to provide richer functionality, flexibility, extensibility, and federation. Rather than a separate system for HVAC, lighting, etc., these are integrated in a common BOS. New systems, such as electrical usage monitoring [13], environmental or CO2 sensing [27], or appliance control [11], are often further integrated with these conventional building subsystems to support new operating modes, such as demand response or demand controlled ventilation.

Over the past few years, building operating systems have converged on an architecture similar to that shown in Figure 1. This has a set of services acting as a hardware presentation layer for a heterogeneous set of hardware accessed over a variety of interfaces. The HPL serves to promote different devices to a common communication format on a single bus. Higher level services, including archivers for time-series data, metadata stores, query processors, controllers, arbiters, and schedulers attach to the bus (as additional persistent processes) and support portable applications, including occupant-centered conditioning, energy analytics, diagnostics, prognostics, model predictive control, and so on. These applications are also essentially persistent processes dropped into the cyberphysical building distributed system, where they can be accessed in turn by services performing data analysis, automation and storage.

Despite the convergence at other layers, there has not been an agreement on how security and authorization are implemented. Some systems implement security in the broker/archiver (e.g., Mortar.io [22], Sensor Andrew [25], HomeOS [15]), some have distinct security models for sensing vs actuation (e.g BOSS [14]), some assume applications are fully trusted (e.g VOLTTRON 2.x [2]) and many simply rely on the applications to implement security.

Here we provide a stronger foundation for authorization within and across administrative domains of the built environment along

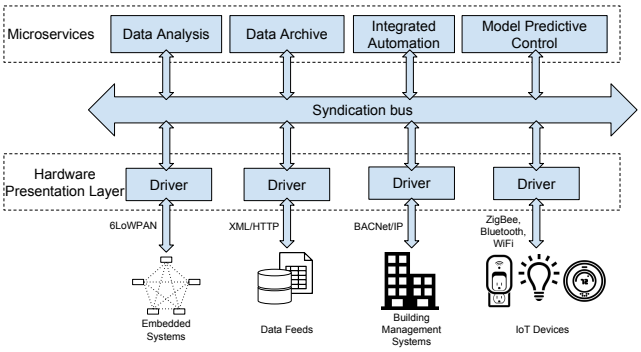

Figure 1: Common Building Operating System Architecture

with DDOS protection of all the components while preserving this converged architecture and its constituent components.

2.1 Hardware Presentation

The many distributed sense and control points in the built environment are on a variety of different interconnects (RS485, BACnet, Ethernet, WiFi, LoWPAN, etc.) with a variety of different access methods and protocols. To unify these, BOS wrap these devices with persistent processes acting as a hardware presentation layer. This layer of abstraction, which centers on the common notion of a persistent process acting as a device driver, has taken a number of similar forms in prior work. For example, HomeOS [15] features service interfaces, Beam [26] has adapter modules, and BOSS [14] has sMAP drivers [13]. In all these cases, the drivers communicate over a common syndication layer for device and service discovery, data reporting, and actuation.

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

2.2 Syndication

The syndication tier is the backbone in Figure 1 that connects all services, drivers and applications. The majority of BOS have converged on the publish/subscribe resource-oriented communication paradigm within this syndication tier as it offers advantages such as location transparency, easy multicasting of messages between arbitrary and dynamic collections of producers and consumers, and a clean abstraction for event-based programming. In BOSS [14] and sMAP [13] the pub/sub broker is strongly coupled with the data archiver, and is used for sensing but not all forms of actuation. Similarly HomeOS [15] and Beam [26] both have a single server performing message dispatch and archiving, although the semantics of the communication channels differ. Sensor Andrew and Mortar.io use XMPP [29] which is closest to Figure 1 as the broker performs purely syndication and authorization, leaving data archival as a distinct service.

Note that in all of the above systems, the syndication mechanism is also a centralized authorization authority, a pattern this work is engineered to avoid.

2.3 Storage

Most building operating systems gather and store sensor data for applications, such as energy data analytics and occupancy-based environmental control. Storage of and access to this data is typically managed by a central authority, normally the building administrators. To deploy a sensor that reports building data or an application that consumes sensor data, one has to obtain permissions from the central authority. This pattern is present in BuildingDepot [1] and in BOSS [14] for example. Some approaches distribute the data (e.g., Mortar.io [10, 22]) but employ a centralized authorization system.

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

2.4 Applications

Much of the functionality of a building operating system is derived from the set of persistent services and applications it hosts. Some of these perform analytics on data to create new streams of information (e.g., anomaly detection [17] and occupancy sensing [3]), some of these perform control of building processes (e.g., demand response [3] and model predictive control [19]), and some provide miscellaneous services (e.g., visualization services).

3 BOSSWAVE

The vast majority of authorization systems used today rely on a connected central authority. If we consider the situation of a participant wanting to show another participant that they are authorized for an action, the prover and recipient must contact the authorization authority to verify each interaction, any changes to authorization must be done via the authority, and the authorization authority must be completely trusted.

For example in XMPP, the server forwarding messages is also the authorization authority (or an external LDAP server). Managing permissions (e.g., to grant permissions to a new device) requires contacting whoever manages that server. If the server is compromised, many new accounts and permissions can be created, or existing ones deleted, thus the server must be trusted.

We have built enterprise solutions in this paradigm for long enough that we no longer think of it as onerous within a single administrative domain, but if we take a step back and consider the natural path of applications and operating systems for the built environment – moving from systems concerning a small number of people within single buildings to large numbers of people in collections of buildings and physical resources owned by different parties – it becomes evident that this model is overly restrictive. Every trust relationship is mediated by some implicit trust of an authority per administrative domain who records it (and could change it at will). There is no singular objective truth but rather a fragmented set of *views of trust* with no guarantee of consistency. Basic delegation requires transitive trust relationships, but at present it is a prohibitively complex and manual process to construct these across administrative boundaries. Confidence in the security of the system as a whole requires complete trust of so many distinct authorities that it could never be practically attained.

Recently, advances in cryptography and distributed computing in the form of blockchains have offered a game changing primitive:

the smart contract. Briefly, this enables a piece of code to manage a piece of globally visible state *without requiring any trust, authority or coordinator.* Applied to the built environment, this means that we can take some logic governing permissions, rules about how they can be granted or revoked, and a table of all the permissions in existence, and put that in a contract on the blockchain. At this point it becomes self-sufficient: there is nobody who can compromise it and nobody who must be trusted for it to continue functioning. Furthermore, the blockchain is *monotonic*. If someone uploads a revocation, she can be sure that it becomes a persistent part of the global state and is not "forgotten" as part of an attack. This guarantee is remarkably hard to achieve and the authors are not aware of any other authorization system providing this.

As an implementation of this idea, BOSSWAVE is a fully democratized and decentralized publish/subscribe communication and authorization platform that uses smart contracts on an Ethereum blockchain to store a consistent global graph of permissions, avoiding the reliance on any centralized authorities.

3.1 WAVE Background

A full treatment of the design and implementation of BOSSWAVE's authorization layer, named WAVE, is beyond the scope of this work and will be described in a dedicated paper. Here, we sketch the primary concepts and their application to the built environment. Traditionally, authorization systems work with principals such as an email address or username. This works because there is an authority that can attest that the individual possesses the given email address. In an authority-less system we need a principal that is self-proving. In WAVE this is an *entity*. Every person, device, service or intermediary (e.g., a security group) will possess an entity. Concretely, it is a keypair identifying a principal that can give permissions, receive permissions and sign messages. An entity does not have an identity, rather it represents whatever or whoever holds the secret part of the keypair.

Authorization concerns controlling access to a set of resources. In typical systems the central authority is the root of permissions; it begins with all permissions and every participant receives them from that root. In an authority-less system we need a similar way to bootstrap permissions but in a democratized manner. The solution is to strongly couple the source of permissions with the resource. In WAVE, a *resource* is identified by a Uniform Resource Identifier (URI). A URI in the World Wide Web begins with a host name, which is the authority for that URI. Similarly, in WAVE a URI begins with a *namespace* which is the root of permissions for that resource.

Any entity can create a namespace and become the root of permissions for it (achieving the democratized goal). No coordinator or authority governs this process. Any URI beginning with the namespace identifier is considered as being within that namespace. Importantly, due to the nature of the cryptography involved, it is impossible for namespaces to collide, so there is always only a single entity as the root of permissions in a namespace possessing complete and unrevokable permissions to all resources within it.

Resources are used for communicating, so we need a mechanism for other entities to obtain permissions on resources. While a traditional access control list maintained by the namespace creator would work, WAVE uses a more powerful mechanism. Instead

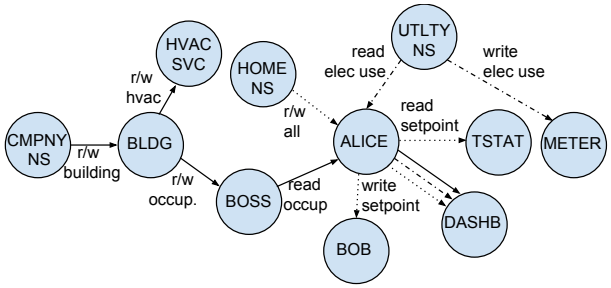

Figure 2: A portion of the delegation of trust graph. Different lined arrows are DoTs on different namespaces

of a list of absolute permissions (if an entity is in the list it has permissions), WAVE uses *delegated* permissions which relates the permissions of one entity to those of another. The primitive here is a *Delegation of Trust* which states that the *granter* gives a *grantee* a set of *permissions* on resources matching a *pattern*. This tuple of granter, grantee, permissions, and pattern is then signed by the granting entity and stored in the contract as part of the globally consistent and visible state. This mechanism has important properties:

(1) The granter and grantee do not communicate (so the grantee can be offline).
(2) No authority or coordinator is involved (not even the namespace creator).
(3) The granter need not have the permission it is granting at the time it creates the DoT, allowing out of order granting
(4) The granting entity can grant permissions on any resource, in any namespace, not just namespaces they create.

In essence this creates *democratized authorization*: all entities having a permission on a resource are equally capable of delegating that permission to others. Note that the ability to delegate is itself a permission, so if required it is possible to grant the ability to interact with a resource, but not the ability to further delegate it. The security and distributed manageability properties of the system are derived from this mechanism.

Proving Authorization. A grant says "you can have this subset of the permissions that I have" so the existence of a DoT giving permissions to a grantee is not sufficient to prove that the grantee has those permissions. A proof for a resource in a namespace must show that there exists a path from the namespace creator (which has unalienable permissions to all the resources) to the proving entity. Furthermore the intersection of the permissions granted by all the traversed DoTs must be greater than or equal to the permissions being proven. Consider Figure 2 showing a portion of the global permissions graph. A company namespace has granted permissions to a building entity. The building entity can now autonomously manage access to all the resources within that building without communicating with the company namespace creator. It grants a subset of permissions to an HVAC service entity and to an employee labeled Boss, who in turn grants an even narrower subset to Alice. Alice's entity has permissions on multiple different namespaces, and she can interact with resources across these namespaces using the same entity.

When Alice wants to interact with a resource (for example observing real-time conference room occupancy) she uses the chain of delegations of trust beginning with the namespace creator and ending with her entity as a proof that her interaction is authorized. This proof is self-standing, objectively correct (no honest party would refute the proof) and can be verified by anyone.

The autonomy that this delegation confers is the key to the *democratization* of authorization in BOSSWAVE. Alice can now delegate the access she has on resources from multiple namespaces to a third party application (e.g., a dashboard) that she wishes to use, and she can do so without requiring anything from the various people and organizations that gave her access.

While simple, this example captures many of the problems that are difficult to solve in existing systems: Alice can delegate to an application some permission to access resources spread across multiple administrative domains and fully manage the lifecycle of this delegation. In a traditional system this would not be possible to do both securely and efficiently; she would either have to convince the subsystem administrators to make multiple accounts for her application on the various authorization servers (an overhead cost), or share her usernames and passwords with the application which then renders Alice and the application indistinguishable (a security cost). In WAVE, this delegation can happen without merging the application and Alice's digital identity, but also without any effort on the part of anyone else. Furthermore this delegation is fully auditable: the entities in the various namespaces are aware that these delegations exist.

Giving the ability to manage delegations to the stakeholders is critical for addressing lifecycles: if Alice uninstalls the application, she can revoke its permissions to resources across a wide set of namespaces without asking the various admins to delete accounts. Furthermore she can be sure that that access is fully gone, a guarantee she would not have had if she shared her password with the application (and implicitly its vendor).

3.2 Syndication tier

While the authorization system described above can work in isolation, to fully solve issues of device protection and resource veracity it is necessary to rework the publish/subscribe syndication mechanism around which BOS are constructed. We have developed BOSSWAVE as a syndication system that integrates the authorization primitives of WAVE.

Coupling between the authorization and syndication tier is done by the namespace creator. After a namespace has been created, it is bound to a *designated router*. This is an entity belonging to a server (perhaps on premises or in the cloud) that will perform syndication on the namespace. The router independently manages the binding from its public key to an IP address and port to contact. Both of these bindings are managed in a contract on the blockchain, similar to the delegations of trust and entities described above.

Communication between devices, services, applications and people is done by *publishing* and *subscribing* to resources. Concretely, the entity wanting to interact with a resource forms a message containing the resource URI, a proof as described above, and an action like subscribe or publish. This message is signed by the entity, providing authentication, and sent to the designated router

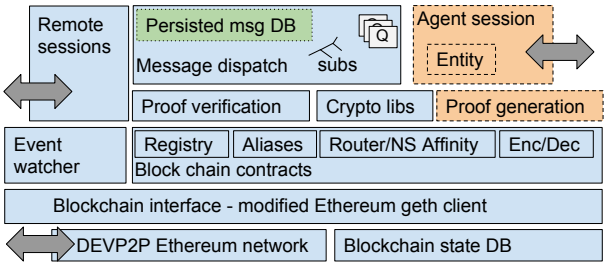

Figure 3: The BOSSWAVE router/agent daemon. Shared parts are solid, dashed boxes are in agents only and the dotted box is in the router only

for the namespace. Because the sender learns the router's key in advance from the blockchain, the session can be secured against eavesdropping and man-in-the-middle attacks without trusting either DNS (which can be poisoned) or SSL certificate authorities (which regularly[1] issue unauthorized certificates "by accident").

The signed, proof-carrying message is opportunistically validated by the router and dropped if unauthorized. While not required for the security model (any recipient will independently validate the message, not trusting the router) this does provide a useful property while the router is uncompromised: unauthorized messages do not get forwarded. In the built environment it is typical for devices to be resource constrained, so they can easily be overwhelmed by unauthorized traffic even if they are aware it is unauthorized. This denial of service (DOS) attack is very common in the modern internet. The magnitude of a layer 7 DOS attack is bounded by the blockchain use in WAVE, and *even at the upper bound*, the impact is negligible (end-to-end latency remains within the standard error), an immensely useful property not present in syndication platforms like XMPP or MQTT.

While a designated router performs a similar role to the broker present in systems like XMPP or MQTT, there are some important differences:

(1) The router is not an authority for the namespace. It cannot grant or revoke permissions
(2) The router is not fully trusted. It cannot create messages associated with resources and it cannot read messages[2]
(3) The designated router for a namespace can be changed without affecting the resources and the services interacting with them

The designated router should be thought of as a service employed by the namespace that can be replaced at will, rather than a piece of trusted infrastructure that must be carefully controlled.

3.3 System implementation

BOSSWAVE is implemented in Go [16] and consists of two daemons: a router which typically runs on a server-class platform with a persistent internet connection, and an agent which runs on every platform using BOSSWAVE resources such as IoT devices, servers, laptops, phones etc. An overview of the software is shown in Figure 3. This software is fully open source and available to download [8].

[1] e.g., http://thehackernews.com/2017/03/google-invalidate-symantec-certs.html
[2] The feasibility of hiding resource contents from the router has been proven theoretically but not implemented in the version of the system presented here.

The authorization tier from Wave is implemented as four smart contracts on an Ethereum blockchain: the **Registry** contract stores the Delegations of Trust, as well as the public parts of Entities (such as their expiry date). DoTs and Entities are revoked by storing the revocation in the contract. The **Alias** contract stores a mapping from a sequence of human readable characters (e.g., "alice19") to a 32 byte blob, usually the public key of an Entity in the registry. The contract ensures these aliases are globally unique and immutable. The **Affinity** contract stores the binding between a namespace and the designated router entity, as well as the binding from the designated router entity to a server's IP address and port. The **Encoding/Decoding** contract is a library used by the other three contracts that can validate Entity, DoT and revocation objects by checking they are well formed and the signature is valid.

To interact with Bosswave an application uses a language binding that connects to the agent (Figure 3 with dashed boxes). As the cryptographic libraries are embedded in the agent, the language bindings are trivial to generate. To allow the agent to perform Bosswave operations on behalf of the application, the entity object is transferred by the application to the agent upon startup. The application can send high level commands like "publish this payload to this resource" and the agent takes care of generating the proof, serializing the message, signing it, resolving and verifying the designated router and transferring the proof-carrying message. Similarly for subscription, the agent verifies every message and only forwards valid messages to the application. As a result of this architecture, applications or services using Bosswave are no more complex than those using legacy syndication (e.g. MQTT or XMPP).

Note well that there is no *blockchain server*. The blockchain exists without any server, authority or coordinator by consensus among the body of clients, all of whom are equal. So while, for example, the registry contract code is acting as an authority, it is autonomous and cannot be tampered with. This is trivializing a highly complex and fascinating system, please consult [28] for more information.

4 XBOS

Bosswave provides security and communication primitives, but by itself does not act as an operating system. XBOS, the eXtensible Building Operating System, is built on top of Bosswave and fulfills the role of a building operating system by using *microservices*: small single-purpose persistent processes. The notion of a microservice architecture for large scale distributed system design is already well known and well proven. The contribution here is the synergy between microservices and the communication and authorization model of Bosswave.

4.1 Containerized, Persistent Services

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

(1) How does a service obtain permissions to consume its input resources and produce its output resources?
(2) How does a service obtain its initial configuration?
(3) How does a service get scheduled to run?
(4) How is the service isolated such that failure does not couple to colocated but unrelated services?
(5) How is a service monitored and how is authority to monitor it obtained?
(6) How is a service retired?

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

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

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

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

(1) What code to run
(2) What environment it needs (libraries and version etc.)
(3) The configuration parameters
(4) The isolation parameters (CPU time, memory, network access, etc.)
(5) Placement restrictions (requiring a particular architecture, or particular network)

The microservice is then instantiated simply by publishing this manifest to a Bosswave URI representing a Spawnpoint instance that satisfies the necessary placement constraints. This instance is responsible for managing a specific collection of computational resources (e.g., an on-premises server or a cloud-based virtual machine). Note that the publication of this manifest is no different from any other Bosswave operation. Upon receipt of a new service manifest, the Spawnpoint daemon builds a docker container with the correct environment and code. It injects the configuration parameters (including the Bosswave Entity that represents

the microservice) and launches the container, using cgroups to enforce the isolation parameters. Various monitoring metrics such as CPU usage, memory usage and program output are streamed to a Bosswave URI representing that microservice. Monitoring and logging of microservices is thus accomplished through Bosswave subscriptions.

This approach borrows heavily from microservice systems such as Kubernetes [18], but is distinguished by the use of Bosswave for the protected control interface (rather than a REST API as in Kubernetes). Spawnpoint makes creating, manipulating, and monitoring a microservice no different from any other operation on a Bosswave URI. Therefore, Spawnpoint inherits all of the permission verification and delegation benefits of Bosswave without any complexity added to its own implementation. In particular, this gives a user the ability to independently and locally delegate partial access to the services they control and manage that delegation. In addition, it adds transparent and effective DDOS protection to services, a problem usually poorly mitigated by costly over-provisioning.

This ability to partially trust services (with fine grained control over what they have access to and what resources they use when running), allows us to run third party applications that we do not fully trust. This is a critical feature provided by computer operating systems for decades, but is often ignored in the built environment context. One could imagine that building monitoring and alerting would be an off-the-shelf third party application that you install on your building. You vaguely trust the application to do its job, but that does not mean you should be forced to trust that it will not attempt unauthorized actions or that you must audit the application yourself (e.g., in VOLTTRON [2]).

4.2 XBOS services and interfaces

While Bosswave permits resource URIs to have any form, we have found it convenient to create a set of conventions on URI format as shown in Figure 4. These conventions then permit service and interface discovery, as well as metadata propagation.

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

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

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

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

Metadata can be attached to services and interfaces by using persisted messages in Bosswave. This metadata provides additional

Figure 4: The XBOS URI structure capturing services and interfaces to enable autodiscovery

context for the service or interface. For example it may convey that the thermostat service represents a device in a particular room in a particular building.

The format of this metadata is not constrained by Bosswave, but again some conventions help with interoperability. Within XBOS we are using the Brick schema [5].

4.3 Types of microservices

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

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

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

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

In the majority of cases, the security policy of the application can be completely expressed as a set of permitted input resources and permitted output resources, e.g., what the application can see and who can see what it produces. In some cases (e.g., in [4]) more complex policy is required, such as "X can see office occupancy only during work hours, or X can see aggregate information about a floor, but not individual offices". This can be implemented by spawning a policy adapter microservice that consumes the raw information, and publishes information in accordance with the policy. The end user is then granted permission to consume only the output of the policy adapter, not the raw data. This allows arbitrarily complex logic to be enforced.

4.3.3 Controllers. A controller consumes a set of sensing and parameter resources and writes to a set of actuation resources. Examples include things like setpoint schedules, setbacks, etc. As

above, there are no placement restrictions, which allows complex controllers such as model predictive control, to be executed where computational resources are cheap.

4.4 Heavyweight services

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

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

5 EVALUATION

The goals of BOSSWAVE are to solve problems of delegation, federation and protection at scale. We have long-lived real-world deployments but only at a modest scale. To fully verify the scalability, we show that the system is capable of handling the load associated with unifying a city-scale built environment under a single syndication and authorization system. Drawing from public land-use and tenancy data for San Francisco, we emulate the static load and churn of changing authorization associated with natural city evolution.

This emulation is performed by executing the same BOSSWAVE commands as would be performed in real use. The emulated entities and permissions are no different from real entities or permissions. This gives us a high degree of confidence that our observations within this emulation are what we expect if a city were to adopt and deploy this system.

The experimental setup for the city-scale emulation is shown in Figure 5. An event-based emulation of nearly a million people and over a hundred thousand buildings draws from the statistical model and issues commands to an agent which acts on them as if they were real commands, putting the created entities and delegations of trust into the global state. Separately, 100 containers running in the cloud emulate participants in BOSSWAVE. To capture the effects of differing internet connections, the `netem` feature of Linux is used to emulate a spectrum of internet speeds and latencies. Bandwidth usage, CPU usage and memory usage statistics are streamed from each container to a central database for analysis.

To set up the emulation, nearly a million distinct entities are created for people living in the city. Then an additional million entities are created for leases (and titles), apartment buildings, apartment owners, and common devices such as thermostats and meters as shown in Table 1. These intermediaries are created as distinct entities to capture the real hierarchy and delegation present in a city. An apartment leasee, for example, obtains permissions to the thermostat resources using a proof traversing their lease, apartment building and then building owner.

In total, roughly 2 million entities and roughly 3 million delegations of trust were created to represent the initial state of San Francisco. Again, due to the authority-less nature of BOSSWAVE it

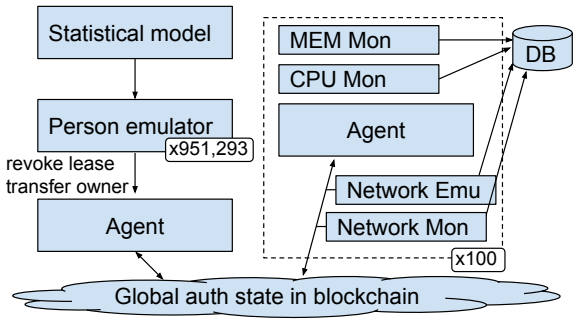

Figure 5: Overview of city-scale emulation

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

Table 1: Number and type of entities and DoTs in the city-scale emulation

was necessary to create this state in the same manner as it would be created in a real deployment – there is no "developer shortcut" as the developers have the same power in the system as everyone else in the world. The emulation then proceeds to change the permissions in the city by following the statistical model of ownership and occupancy change. As apartments and houses change occupancy, old leases or titles are revoked and new leases and titles created. At the peak of business hours, the city emulation causes roughly 150 changes of permission per hour. We separately characterize the capacity of BOSSWAVE at a rate of approximately 4000 changes of permission per hour, although this limit is driven by the blockchain and changes as the body of clients vote on what the bandwidth of the chain should be.

5.1 Agent Resource Usage

We have measured the cost of participating in BOSSWAVE, which requires synchronization with the WAVE blockchain. For space reasons, the full results will be presented in a separate paper. The cost of participation under normal conditions is reasonable: a constant CPU load of about 5% of a desktop-class processor core or 50% of an embedded-class ARM core. There is also an associated constant network bandwidth use of between 2KB/s and 12KB/s. However, this CPU and network overhead is still significantly higher than the cost of interacting with centralized authorization services and may pose a problem for battery powered or highly-constrained systems.

One solution to this is the *remote agent*. This takes the expensive parts of the BOSSWAVE agent – blockchain participation, proof building and message verification – and offloads it to a separate trusted platform. The trust relationship between the local agent and

the remote agent is established out of band. The local agent then requires minimal resources. As an example, we have thousands of resources associated with ARM Cortex-M0+ class wireless sensors. These cannot participate in the blockchain directly, but can perform the symmetric encryption required to communicate with a remote agent that has the blockchain and will do the work of building proofs as well as verifying and signing messages. The symmetric keys for establishing trust are installed at commissioning time. While this does mean the remote agent must be trusted, this pattern is still very different from a centralized authority because anyone can create their own remote agent, so no requirement for trust of a third party is introduced.

5.2 Syndication performance

The signatures on every message are the most significant term in syndication performance. On server class devices the impact is minimal, adding approximately 75 μs to message routing time. On embedded devices such as a Raspberry Pi the signatures take longer to generate and verify, approximately 1 ms and 1.5 ms respectively. This leads to a throughput of roughly 600 msgs/s per core. In practice, the situations where such embedded class devices are used (e.g., as a local gateway to a BMS or sensor network) rarely have such high message rates, so the cost of the cryptography is not onerous. On server class Bosswave router performance is comparable to popular syndication systems, easily capable of routing thousands of messages per second.

The cost of generating a proof is more variable, and depends on how many DoTs lie between the namespace entity and the prover. For complex namespaces involving thousands of entities this can take hundreds of milliseconds on an embedded device, but in practice the proof is cached and remains valid until the permission graph changes along the path used (due to revocation or expiry). We expect this to occur infrequently, maybe once every few weeks, so the vast majority of resource interactions use a cached proof.

5.3 Device protection

In many existing building operating system architectures, the traffic appearing at an end device is dynamic — it changes based on who is actuating it (e.g., sMAP [13]) or consuming its sensor feeds (e.g., Mortar.io [22]). This makes it difficult to configure layer 3 firewalls to block illegitimate traffic upstream. In Bosswave the device and its agent only speaks to the designated router. As the parties authorized to control the device or consume its data changes, the IP address speaking to the device remains the same. While simple, this difference in network architecture makes it much easier to protect the device: only allow traffic to and from the designated router. At layer 7, the designated router ensures that unauthorized traffic does not get forwarded to the device. The problem becomes reduced to hardening the designated router against denial of service attacks. This task is also made easier by Bosswave as every message is signed, tying it to a specific entity. If the entity is misbehaving, it is blacklisted, causing future messages from that entity to be dropped. As creating new entities requires interacting with the blockchain, a Sybil attack where new identities are created to circumvent the blacklist is not possible — it is trivial for the designated router to blacklist entities faster than they can be created.

5.4 Deployments

Deployment	Length	Device Type	# Resources
Campus Building	9 Months	Thermostats	21
		WSN Mote	64
		Power Meter	5
Residential House	13 Months	Thermostats	84
		WSN Mote	300
		Power Meter	12
Research Lab	9 Months	WSN Mote	1485
Air Velocity	7 Months	Sensor	12

Table 2: Summary of the resources associated with HPL microservices in three deployments

Aside from the city-scale emulation, we also present details about four deployments of Bosswave and XBOS, as summarized in Table 2. Together they have securely handled more than a billion resource interactions (involving proof generation and verification) over thousands of resources. The experimental air velocity sensor publishes hundreds of raw readings per second and a service on a Spawnpoint in the cloud performs a suite of analytics to convert those to useful data which are then subscribed to by research collaborators. Table 2 summarizes the length of deployments, type of devices and number of resources. At the time of writing we are deploying this system in 20 small to medium commercial buildings.

The residential installation of XBOS features a Spawnpoint instance deployed on an on-premises PC hosting containerized microservices that back various devices and services. Thus, the devices in the home are monitored and controlled purely through Bosswave. This has a number of security benefits. For example, the smart plug controlling the electric vehicle charger is the TP-Link HS-110, which communicates using a network protocol that can be trivially compromised.[3] By firewalling the device so that it can speak only with the Spawnpoint-hosted service that acts as its Bosswave proxy, only authorized traffic appears at the device.

6 SUMMARY

In summary, this work builds upon the established pattern of a building operating system – microservices performing hardware presentation, analytics and data archival linked together by syndication – published in work such as BOSS [14], Sensor Andrew [25], Mortar.io [22], etc. We affirm this pattern, and solve four outstanding problems that arise as at city-scale. These are the delegation of permissions, federation across multiple administrative domains, protection of devices, and execution of microservices.

These problems have been identified in isolation in work such as SensorAct [4] which identifies the need for stakeholders to delegate how their own data is accessed, but a solution that solves these problems without a central authority has thus far not existed. Without this property, the transition from a building operating system to an operating system for the built environment comprising millions of administrative domains will be mired in management overhead stemming from exponentially increasing distinct views of trust.

We present the first system to provide strongly consistent global authorization without a central authority, solving issues of delegation and federation. By coupling this in a new publish/subscribe

[3] https://www.softscheck.com/en/reverse-engineering-tp-link-hs110/

mechanism we realize a syndication tier that offers strong guarantees on message authenticity and privacy. Notably, the end device only ever receives authorized traffic and only from a single host, resolving issues of device protection. Unlike existing work such as [14, 22], the service routing messages cannot change permissions, forge messages or reveal resources to unauthorized parties.

The final problem of microservice execution is cleanly resolved by combining the authorization and syndication mechanisms with a container execution environment which provides isolation, monitoring and administration of persistent processes while preserving BOSSWAVE's delegation and federation properties.

The system is fully implemented and deployed at a global scale. We anticipate researchers will design and deploy interoperable microservices using this system going forward, following examples such as XBOS [9] and leveraging the extensive libraries created for this purpose [6, 7]. An examination of recent work published in BuildSys shows that many systems are already designed around a compatible syndication pattern and could leverage BOSSWAVE with minimal effort.

7 CONCLUSION

We present a natural extension to the design of building operating systems that leverages an authority-free syndication layer to enable operation at city-scale. This unifying system solves problems of delegation, both within and across administrative domains, and of federation which allows applications to span the built environment. Extending this to a secure syndication tier efficiently solves issues of device protection and DDOS that plague the modern Internet. Furthermore, building this into the syndication tier makes the system practical to deploy: the syndication tier is already the unifying element, and we can realize all the authorization and protection benefits while still providing the near-universal publish/subscribe API. Resource-oriented rules allow natural and transparent expression of many application security policies, so the application need not be aware of the system to benefit from it. This serves both to reduce new application complexity and to allow porting of existing applications to BOSSWAVE. With these tools, we can move from the building to the built environment securely and efficiently.

ACKNOWLEDGMENTS

This work is sponsored in part by the California Energy Commission, Department of Energy grant DE-EE0007685, the Fulbright Scholarship Program, and National Science Foundation grant CPS-1239552. We also gratefully acknowledge the assistance of Gabe Fierro and Raluca Ada Popa.

REFERENCES

[1] Yuvraj Agarwal, Rajesh Gupta, Daisuke Komaki, and Thomas Weng. 2012. Buildingdepot: an extensible and distributed architecture for building data storage, access and sharing. In *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*. ACM, 64–71.

[2] Bora Akyol, Jereme Haack, Brandon Carpenter, Selim Ciraci, Maria Vlachopoulou, and Cody Tews. 2012. Volttron: An agent execution platform for the electric power system. In *Third international workshop on agent technologies for energy systems valencia, spain.*

[3] Omid Ardakanian, Arka Bhattacharya, and David Culler. 2016. Non-intrusive techniques for establishing occupancy related energy savings in commercial buildings. In *Proceedings of the 3rd ACM International Conference on Systems for Energy-Efficient Built Environments*. ACM, 21–30.

[4] Pandarasamy Arjunan, Nipun Batra, Haksoo Choi, Amarjeet Singh, Pushpendra Singh, and Mani B Srivastava. 2012. SensorAct: a privacy and security aware federated middleware for building management. In *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*. ACM, 80–87.

[5] Bharathan Balaji, Arka Bhattacharya, Gabriel Fierro, Jingkun Gao, Joshua Gluck, Dezhi Hong, Aslak Johansen, Jason Koh, Joern Ploennigs, Yuvraj Agarwal, et al. 2016. Brick: Towards a unified metadata schema for buildings. In *Proceedings of the ACM International Conference on Embedded Systems for Energy-Efficient Built Environments (BuildSys). ACM.*

[6] UC Berkeley. 2017. BOSSWAVE Golang library. (2017). https://github.com/immesys/bw2bind

[7] UC Berkeley. 2017. BOSSWAVE Python library. (2017). https://github.com/SoftwareDefinedBuildings/bw2python

[8] UC Berkeley. 2017. BOSSWAVE source code. (2017). https://github.com/immesys/bw2

[9] UC Berkeley. 2017. XBOS documentation. (2017). https://docs.xbos.io/

[10] M. Buevich, A. Wright, R. Sargent, and A. Rowe. 2013. Respawn: A Distributed Multi-resolution Time-Series Datastore. In *2013 IEEE 34th Real-Time Systems Symposium*. 288–297. https://doi.org/10.1109/RTSS.2013.36

[11] Kaifei Chen, Takeshi Mochida, Jonathan Fürst, John Kolb, David E. Culler, and Randy H. Katz. 2016. *CellMate: A Responsive and Accurate Vision-based Appliance Identification System.* Technical Report UCB/EECS-2016-154. EECS Department, University of California, Berkeley.

[12] Ang Cui and Salvatore J Stolfo. 2010. A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 97–106.

[13] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz, and David Culler. 2010. sMAP: a simple measurement and actuation profile for physical information. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*. ACM, 197–210.

[14] Stephen Dawson-Haggerty, Andrew Krioukov, Jay Taneja, Sagar Karandikar, Gabe Fierro, Nikita Kitaev, and David E Culler. 2013. BOSS: Building Operating System Services.. In *NSDI*, Vol. 13. 443–458.

[15] Colin Dixon, Ratul Mahajan, Sharad Agarwal, AJ Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. 2012. An operating system for the home. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 25–25.

[16] Alan AA Donovan and Brian W Kernighan. 2015. *The Go programming language*. Addison-Wesley Professional.

[17] Romain Fontugne, Jorge Ortiz, Nicolas Tremblay, Pierre Borgnat, Patrick Flandrin, Kensuke Fukuda, David Culler, and Hiroshi Esaki. 2013. Strip, bind, and search: a method for identifying abnormal energy consumption in buildings. In *Proceedings of the 12th international conference on Information processing in sensor networks*. ACM, 129–140.

[18] The Linux Foundation. 2017. Kubernetes. (2017). https://kubernetes.io

[19] Rasmus Halvgaard, Niels Kjølstad Poulsen, Henrik Madsen, and John Bagterp Jørgensen. 2012. Economic model predictive control for building climate control in a smart grid. In *Innovative Smart Grid Technologies (ISGT), 2012 IEEE PES*. IEEE.

[20] Tridium Inc. 2017. Niagara 4. (2017). https://www.tridium.com/products-services/niagara4

[21] Andrew Krioukov, Gabe Fierro, Nikita Kitaev, and David Culler. 2012. Building application stack (BAS). In *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*. ACM, 72–79.

[22] Christopher Palmer, Patrick Lazik, Maxim Buevich, Jingkun Gao, Mario Berges, and Anthony Rowe. 2014. Mortar. io: Open Source Building Automation System. In *BuildSys-ACM Int. Conf. on Embedded Systems for Energy-Efficient Built Environments*. 204–205.

[23] Manisa Pipattanasomporn, M Kuzlu, W Khamphanchai, A Saha, K Rathinavel, and S Rahman. 2015. BEMOSS: An agent platform to facilitate grid-interactive building operation with IoT devices. In *Innovative Smart Grid Technologies-Asia (ISGT ASIA), 2015 IEEE*. IEEE, 1–6.

[24] David R Raymond and Scott F Midkiff. 2008. Denial-of-service in wireless sensor networks: Attacks and defenses. *IEEE Pervasive Computing* 7, 1 (2008).

[25] Anthony Rowe, Mario E Berges, Gaurav Bhatia, Ethan Goldman, Ragunathan Rajkumar, James H Garrett, José MF Moura, and Lucio Soibelman. 2011. Sensor Andrew: Large-scale campus-wide sensing and actuation. *IBM Journal of Research and Development* 55, 1.2 (2011), 6–1.

[26] Chenguang Shen, Rayman Preet Singh, Amar Phanishayee, Aman Kansal, and Ratul Mahajan. 2016. Beam: ending monolithic applications for connected devices. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 143–157.

[27] Jay Taneja, Andrew Krioukov, Stephen Dawson-Haggerty, and David Culler. 2013. Enabling advanced environmental conditioning with a building application stack. In *Green Computing Conference (IGCC), 2013 International*. IEEE, 1–10.

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

[29] XMPP Standards Foundation. 2017. XMPP. https://xmpp.org. (2017).