Democratizing Authority in the Built Environment

MICHAEL P. ANDERSEN, JOHN KOLB, KAIFEI CHEN, GABE FIERRO,
DAVID E. CULLER, and RANDY KATZ, University of California, Berkeley

17

Operating systems and applications in the built environment have relied upon central authorization and management mechanisms that 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.
This is an extension of work presented in Ref. [3].

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

Additional Key Words and Phrases: Built environment, syndication, microservices, federation

ACM Reference format:
Michael P. Andersen, John Kolb, Kaifei Chen, Gabe Fierro, David E. Culler, and Randy Katz. 2018. Democratizing Authority in the Built Environment. *ACM Trans. Sen. Netw.* 14, 3–4, Article 17 (December 2018), 26 pages.
https://doi.org/10.1145/3199665

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 [25], sMAP [16], BuildingDepot [1], BOSS [17], Mortar.io [26], BEMOSS [27], Niagara [23], 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

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.
Authors' addresses: M. P. Andersen, J. Kolb, K. Chen, G. Fierro, D. E. Culler, and R. Katz, University of California, Berkeley, Soda Hall, Berkeley, CA 94720; emails: {m.andersen, jkolb, kaifei, gtfierro, culler, randy}@cs.berkeley.edu.

1550-4859/2018/12-ART17 $15.00
https://doi.org/10.1145/3199665

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, and 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 [15, 28] 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 Virtual Private Network (VPN) to cross Local Area Network (LAN).

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 article), 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 [33], 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), 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 [16], environmental or CO2 sensing [32], or appliance control [14], 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,

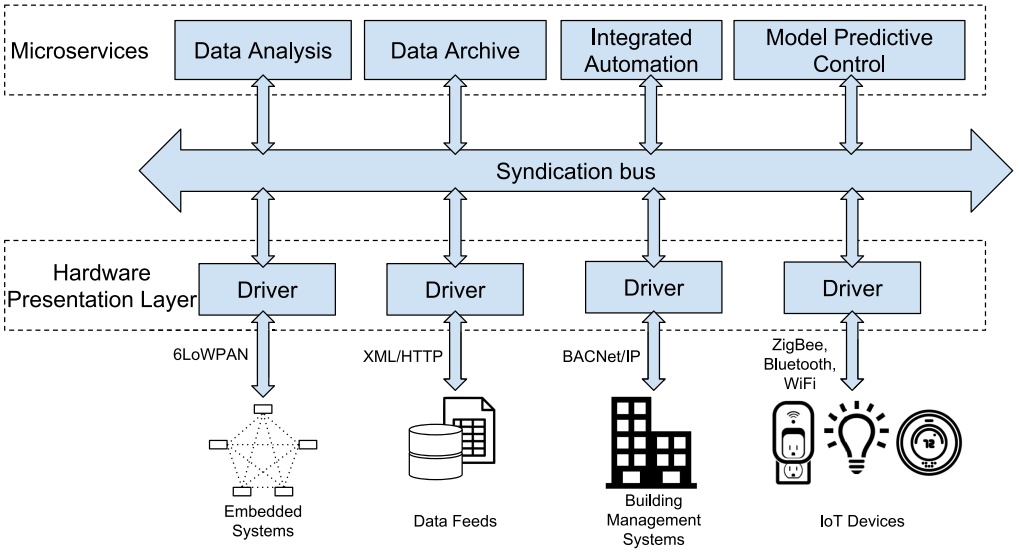


Fig. 1. Common building operating system architecture.

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

Despite the convergence at other layers, there has not been an agreement on how security and authorization are implemented. Some systems implement security in the broker/archiver (e.g., Mortar.io [26], Sensor Andrew [29], HomeOS [18]), some have distinct security models for sensing vs. actuation (e.g., BOSS [17]), 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 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 [18] features service interfaces, Beam [30] has adapter modules, and BOSS [17] has sMAP drivers [16]. 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 [17] and sMAP [16], the pub/sub broker is strongly coupled with the data archiver, and is used for sensing but not all forms of actuation. Similarly, HomeOS [18] and Beam [30] 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 [34], 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 [17], for example. Some approaches distribute the data (e.g., Mortar.io [13, 26]) 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 [20] and occupancy sensing [5]), some of these perform control of building processes (e.g., demand response [5] and model predictive control [22]), 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 is described in a dedicated paper [4]. 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

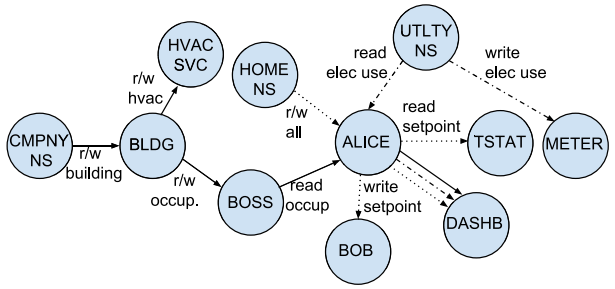


Fig. 2. A portion of the delegation of trust graph. Different lined arrows are DoTs on different namespaces.

would work, Wave uses a more powerful mechanism. Instead 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 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 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

[1] e.g., http://thehackernews.com/2017/03/google-invalidate-symantec-certs.html.

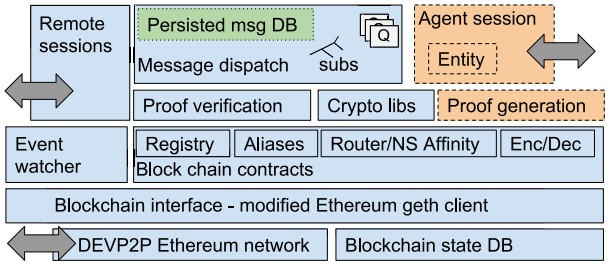


Fig. 3. The BOSSWAVE router/agent daemon. Shared parts are solid, dashed boxes in agents only and the dotted box is in the router only.

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 Extensible Messaging and Presence Protocol (XMPP) or Message Queuing Telemetry Transport (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 [19] and consists of two daemons: a router that typically runs on a server-class platform with a persistent internet connection, and an agent that runs on every platform using BOSSWAVE resources such as IoT devices, servers, laptops, phones, and the like. An overview of the software is shown in Figure 3. This software is fully open source and available to download [11].

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 that 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

[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.

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 Ref. [33] for more information.

4 SPAWNPOINT

BOSSWAVE provides security and communication primitives, but does not deal with the deployment and management of the distributed software that uses these primitives. Distributed systems are often composed of *microservices*: small, single-purpose persistent processes. XBOS, the eXtensible Building Operating System (described in Section 5) is one such example. An oft overlooked issue in microservice architectures, and in building operating systems, in general, is the question of where these services run and how they are 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 Sure, Secure Shell (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, fully detailed in Ref. [24], is a ground-up solution to these problems. Its purpose is to deploy, run, and monitor microservices, which together form a distributed system such as a building operating system, in managed Docker execution containers. Thus, Spawnpoint can be thought of as the "proto-service" combining Docker and BOSSWAVE upon which all other microservices are built. Note that the notion of a microservice architecture for a 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.

Spawnpoint begins with the realization that the resources required for persistent processes (e.g., CPU time, memory) are the same as any other resources (e.g., sensor data) and can be managed using the same Bosswave primitives. Its core component is a daemon process, spawnd, that is responsible for encapsulating a specific collection of computational resources (e.g., an on-premises server or a cloud-based virtual machine) and making it available for use by microservices. Each host running spawnd also runs Docker for local management of its containers, and spawnd wraps the Docker daemon in a secure Bosswave interface. The ability to deploy and manipulate microservices as containers 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. This elevates Docker containers to the status of first-class citizens on Bosswave's message bus.

This approach borrows heavily from microservice systems such as Kubernetes [21], 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]).

The design of the Spawnpoint daemon is shown in Figure 4. Its primary components are:

(1) A Bosswave client to accept remote commands for microservice deployment and control, to publish messages advertising the host's resources for consumption by prospective microservices, and to publish status heartbeats for all containers executing on the host
(2) A Docker client that interfaces with the local Docker daemon to instantiate new services within containers and to manipulate running containers
(3) Global pools of CPU and memory resources that are allocated in slices by Spawnpoint to each running container
(4) A module to periodically snapshot the daemon's state (e.g., the status of each microservice, resource allocations, configurations, and so on) for clean recovery from daemon failures
(5) A container monitor and controller that reacts as necessary to changes in container state (such as failures) and maintains the consistency of resource allocations

4.1 Service Instantiation

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

(1) What code to run
(2) What environment it needs (libraries and version, etc.)
(3) The configuration parameters

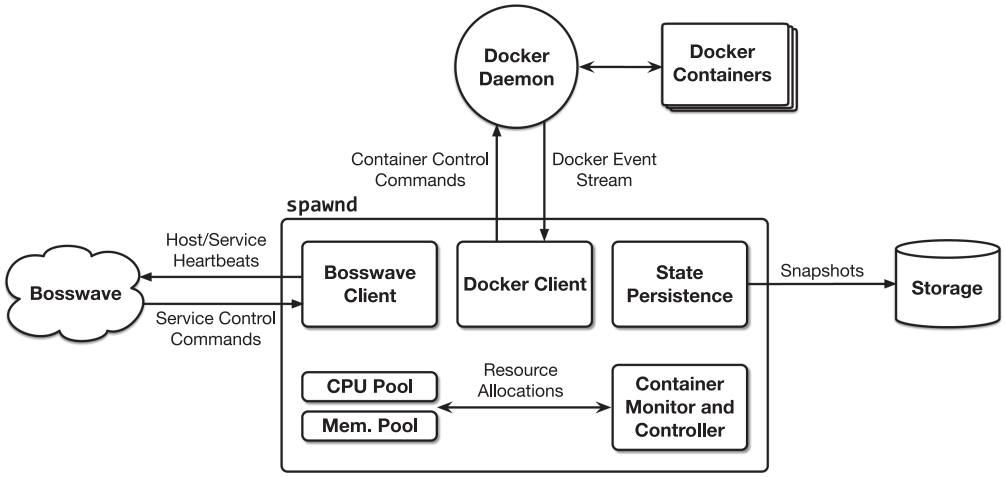


Fig. 4. High-level design of the Spawnpoint host management daemon.

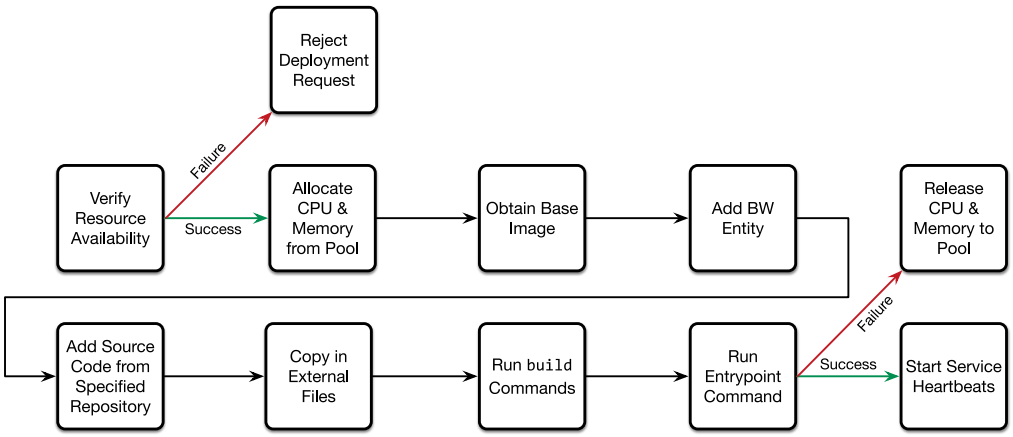


Fig. 5. The process for instantiating a new service on Spawnpoint.

(4) Resource requirements and 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. Note that the publication of this manifest is no different from any other BOSSWAVE operation, allowing any software that can communicate via BOSSWAVE to act as a Spawnpoint client.

Upon receipt of a new service manifest, the Spawnpoint daemon follows the process diagrammed in Figure 5:

(1) Verify that there are sufficient resources in the CPU and memory pools to accommodate the new container. If so, allocate appropriately-sized slices from these pools to the new service.
(2) Create and launch a Docker container in accordance with the manifest's specifications, ensuring that the service will have the correct execution environment and code.

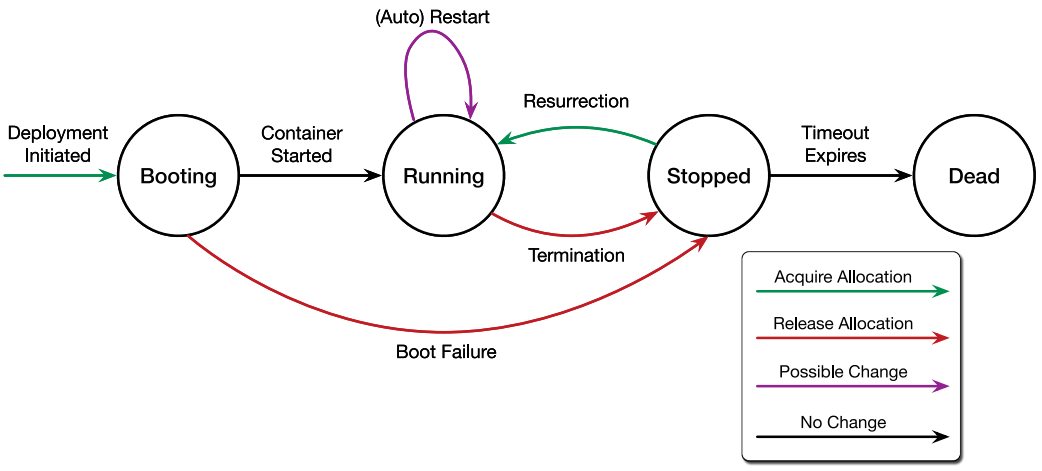


Fig. 6. A state machine representation of a container's lifecycle on Spawnpoint.

(3) Inject the configuration parameters (including the BOSSWAVE Entity that represents the microservice) and any necessary external data or source code files
(4) Launch the container and run any preliminary commands needed to initialize the environment
(5) Execute the microservice using the specified entrypoint command

If a failure occurs at any point during the service boot process, the associated resource allocations are relinquished. During service execution, 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.

4.2 Service Lifecycle

The lifecycle of a Spawnpoint container forms a finite state machine, shown in Figure 6. By monitoring events emmitted by the Docker daemon, Spawnpoint is made aware when a container transitions to a new state and can then modify the associated resource allocations. When Spawnpoint deploys a new service, it first must take the time to complete the container creation process described above. Once a running container terminates, the rest of its lifecycle depends on how the service has been configured. If automatic restarts are enabled, the container will be restarted by Spawnpoint without any user intervention. This continues until an external user or service sends an explicit request to stop the container. After a container enters the "Stopped" state, either because its underlying process has terminated or in response to a request, it becomes a "zombie"—it is neither dead nor alive. If Spawnpoint receives a request to restart the container, it is brought back up with the same configuration, and there is no need to go through the time-consuming container creation process a second time. However, if a timeout expires (the duration of which is a configuration parameter of the daemon), then the container becomes dead. Spawnpoint purges its configuration, and the service must be explicitly deployed again from scratch.

4.3 Service Resource Isolation

Through its use of cgroups, Docker can place limits on container memory and CPU consumption. This prevents containers with faulty code from monopolizing the host's resources. The

Spawnpoint daemon uses this mechanism to allocate resources to individual containers while ensuring that the host does not become overburdened by the containers in aggregate. For both memory and CPU resources, microservice containers are allocated a portion of a shared pool maintained by the Spawnpoint daemon, and these resource allocations are acquired, released, and modified in reaction to changes in container state as shown in Figure 6.

Spawnpoint's approach to the management of the shared memory and CPU pools is relatively simple, but it is easy to reason about and can easily be swapped out for more sophisticated approaches in future versions of the system. Spawnpoint takes a conservative approach to admission control for new services. When a new service is submitted for execution, its configuration must state its resource requirements upfront, and Spawnpoint immediately allocates those resources to the new service from the global CPU and memory pools. If either pool contains insufficient resources, then the service deployment is rejected, and it will not be allowed to run on the host. This means that a service can reserve resources and rest assured that these reservations will be honored, but the service is also prevented from exceeding its allocations to the detriment of co-located services.

4.4 Fault Tolerance

Spawnpoint's use of Docker decouples spawnd from the services that it manages. This is because service containers are independently executing processes that can continue running without Spawnpoint present—they simply won't be managed or monitored. As a result, the Spawnpoint daemon can fail without affecting services running on the host. In this situation, the daemon also recovers gracefully and cleanly reassumes ownership of the containers it is tasked with managing, restarting any services that may have failed in the meantime. To enable this, the Spawnpoint daemon periodically saves a snapshot of its current state to persistent storage. When it resumes execution, spawnd consults this snapshot and restores the necessary service resource allocations.

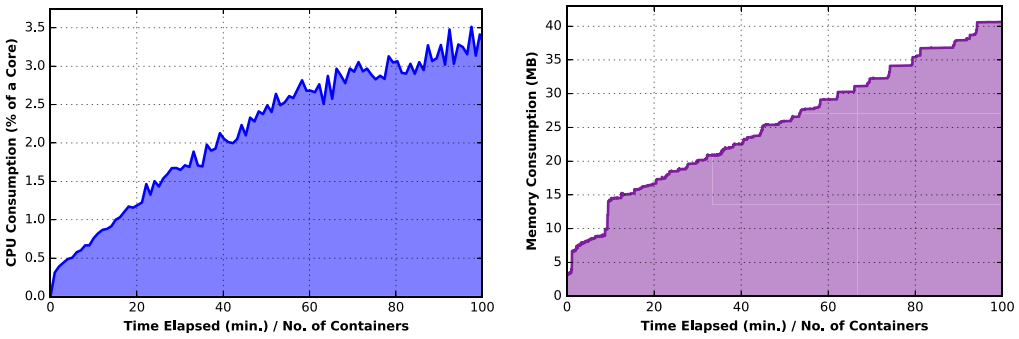
Services running on Spawnpoint may also fail. Spawnpoint is notified whenever a running service terminates and, if specified by the service's configuration, automatically restarts that service's container. These automatic restarts help to maintain the availability of critical services like device drivers. A service developer may also specify a delay period to occur before an automatic restart to avoid problematic crash-restart loops.

4.5 System Performance

We performed experiments to verify that Spawnpoint's implementation meets two important goals:

(1) spawnd is lightweight, allowing the vast majority of its host's computational resources to be devoted to running microservices rather than managing these services.
(2) Spawnpoint's container management imposes only a small overhead on top of native Docker containers.

To evaluate the first point, we measured the CPU consumption, memory footprint, and network traffic incurred by spawnd as it was subjected to a synthetically generated load that increases over time. Note that the system load associated with spawnd is decoupled from the load associated with the services it manages. That is, the cost of managing a service is independent of the cost of executing that service. This experiment was performed on a desktop PC with an Intel Core i7-6700 CPU featuring eight logical cores running at 3.40GHz and 32GB of RAM. The machine ran Ubuntu 16.04 with Docker version 17.05.0 as its container engine. One hundred instances of a simple service were deployed on the the Spawnpoint instance, at a rate of one service per minute.



(a) CPU Consumption (b) Memory Footprint

Fig. 7. CPU and memory demands of spawnd under increasing load.

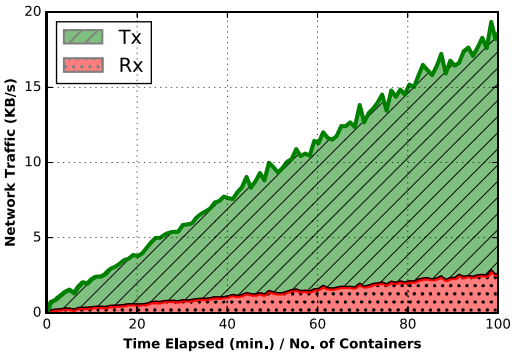


Fig. 8. Network traffic induced by spawnd under increasing load.

As shown in Figure 7(a), spawnd's CPU demands increase as it is tasked with managing more containers, although the rate of increase begins to slow as the total number of containers grows. Even when managing 100 independent containers, the Spawnpoint daemon's CPU requirements peak at a very modest 3.5% of one core. The daemon's memory demands exhibit similar behavior, reaching a peak of only 40MB when 100 services are being managed. The memory curve's resemblance to a step function is a result of the fact that the daemon is implemented in Go. The shape of the graph is an artifact of the Go runtime's memory allocation system. Finally, Figure 8 shows that spawnd's network traffic increases linearly with the number of containers under management but stays under 20KB/sec. combined send and receive bandwidth even when all 100 services are up and running.

To evaluate the overhead of managed Spawnpoint containers versus native Docker containers, we measured the latency involved in carrying out four basic container operations using spawnd and using Docker directly:

(1) Stopping an existing service container
(2) Restarting a recently stopped container
(3) Restarting a running container
(4) Deploying a new service container

Each operation was repeatedly performed on the same hardware described above. Timing measurements were observed entirely within spawnd and, therefore, do not capture any of the network

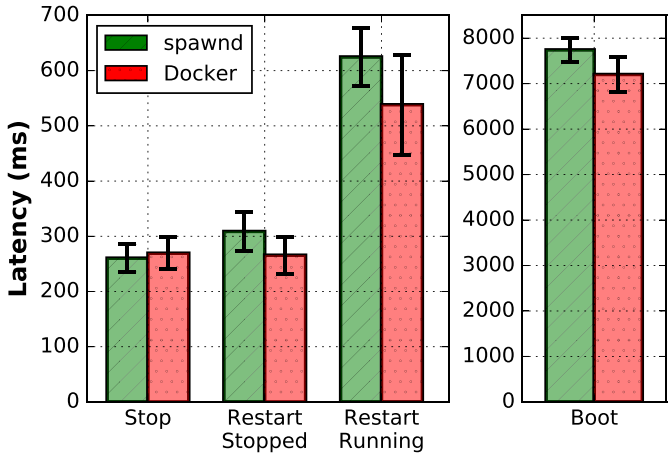


Fig. 9. Latency for operations on both managed and unmanaged containers.

latency associated with sending and receiving the requisite Bosswave messages. However, these measurements do capture the time required to parse each Bosswave message received by spawnd as well as the time required to extract and deserialize the contents of this message. Note, also, that a new service deployment involves downloading and constructing a fresh Docker image for the service's container. This involves significant network communication and makes this operation significantly more expensive than the others.

Figure 9 summarizes the results of the experiment. The height of each bar represents the average latency of a given operation, while the error bars represent standard deviation. The cost of stopping a container is roughly identical for both managed and unmanaged containers. For restarting a container, either running or stopped, as well as booting a new container, latency is lower for unmanaged containers, but the increase in latency for managed containers is relatively small, and the cost of the overall operation remains reasonable when using spawnd. Moreover, the increased latency of managed container operations makes sense given the fact that spawnd has to perform additional work, like resource management and admission control, on top of the underlying Docker container operations.

5 XBOS

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

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

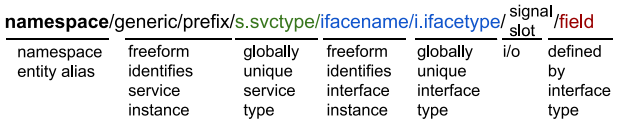


Fig. 10. The XBOS URI structure capturing services and interfaces to enable autodiscovery.

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

5.1 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 10. 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 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 [7].

5.2 Types of Microservices

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

5.2.1 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.

5.2.2 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 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 Ref. [6]), 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.

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

5.3 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.

6 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 11. 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.

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

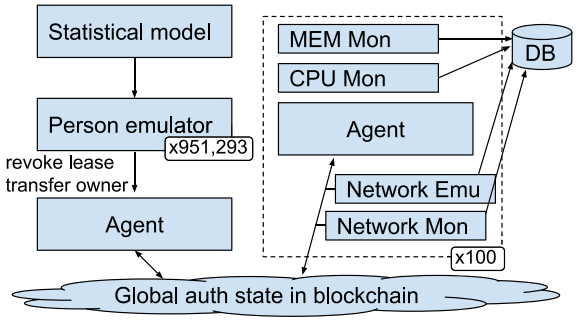


Fig. 11. Overview of city-scale emulation.

Table 1. Number and Type of Entities and DoTs in the 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

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 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.

6.1 Agent Resource Usage

A primary concern in using a blockchain is the cost of an agent participating in the chain. We use low-level instrumentation of the 100 Bosswave nodes performing the emulation and observe them under the different levels of blockchain load caused by the emulation. This allows us to record the impact of BOSSWAVE use on these nodes in terms of CPU, memory, and network bandwidth consumption. We also monitor the current "age" of the chain on each Bosswave node. This is a measure of consistency specific to blockchain-based systems and is important; if an agent cannot keep up with chain updates, it must choose to drop messages or elevate its trust in the router because it is potentially unaware of recent revocations.

Table 2. Breakdown of AMD64 Cloud Nodes in Testbed

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

Experimental Setup. We run BOSSWAVE on three different platforms. Docker containers are used to encapsulate the 100 nodes, consisting of both miners and agents, on Amazon EC2. We use seven `m4.16xlarge` instances featuring Intel Xeon CPUs and Sure, Solid State Drive (SSD)-backed storage that guarantee a baseline performance of about 1,000 Input/Output Operations Per Second (IOPS). We randomly distribute the containers among these hosts with the constraint that no host runs more than 16 containers. This ensures that each host has at least 4 CPU cores and 8GB of memory per container. We use `netem` to shape each container's network latency and bandwidth. A container is thus assigned to one of four networking classes based on a profile of wired Internet connections in North America [8]. The breakdown of nodes among these different constraint classes is given in Table 2. In addition to the AMD64 cloud nodes, three agents are run on Raspberry Pis with quad-core armv7l CPUs and 1GB of RAM, and three agents are run on i386 machines with Intel Atom CPUs and 2GB of RAM.

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

(1) *Fast Sync* [31] occurs when a new node is brought online. The node must synchronize with the block chain to reach a consensus on BOSSWAVE's current state.
(2) *Idle* is a period of minimal blockchain activity. Although blocks contain no transactions, new blocks are mined. Agents must process these new blocks to remain synchronized with the chain.
(3) *Attack* is a period of intense activity, effectively at the level of a concerted attack on the chain by a party with infinite funds. This is compounded by BOSSWAVE's blocks becoming temporarily enlarged, accommodating more than 130 registrations, meaning each block consumes 25M gas, an order of magnitude more than the gas limit of Ethereum's main chain (roughly 4M)
(4) *Normal* is a period of typical city-scale load on the blockchain. The generation of this load is described above.

CPU. Table 3 contains a summary of the CPU measurements collected for all platforms. No BOSSWAVE node was CPU-bound at any point. Miners use multiple CPU cores regardless of the load on the chain, as expected. CPU consumption for regular agents was higher when the blockchain was under attack, but never exceeded $1/2$ a core.

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

Network Bandwidth. The bandwidth differences across Internet speeds are not statistically significant when the agent is caught up on the chain. UDP bandwidth used for peer discovery is not

Table 3. CPU Utilization as [Number of Cores] for Block Chain Participation

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

The first column indicates the role (Agent or Miner), quantity, and peer count and architecture for each node type.

Table 4. Bandwidth [KiB/s] for Block Chain Participation

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

The first column indicates the role (Agent or Miner), quantity, and peer count for each node type.

significant compared to the bandwidth used for transferring state, and is therefore omitted. Table 4 shows the average bandwidth used by agents at different levels of chain activity.

The bandwidth used to participate in the chain is reasonable even during periods of excessive load. That bandwidth can be controlled by limiting the number of peers is likely indicative of an implementation flaw in the Ethereum golang client, as having more peers should not cause a participant to download significantly more data. All downstream traffic difference at attack load between the 20-peer case and the 2-peer case is redundant, so at least 100KiB/s of 114KiB/s is data that need not be fetched from peers. Nevertheless, participation in a chain that is for all intents and purposes under a DoS attack only requires a modest amount of bandwidth.

Chain Age. Chain age is the number of blocks separating the head of the blockchain from the latest block known to a node. Nodes stay up to date on the chain during the idle and normal period, independent of peer count and available network bandwidth. There are short and intermittent periods where nodes fall behind the head of the chain. We never observe a node falling behind by more than nine blocks, equivalent to maximum staleness of about 2 minutes.

Figure 12 shows the age of the chain for each of the 100 EC2-based Bosswave nodes across our emulated attack. All nodes are caught up to the head of the chain until chain activity dramatically increases at 04:30 UTC, whereupon most nodes fall one or two blocks behind but quickly recover. A handful fall more behind and require about 30 minutes to recover. We found, upon further investigation, that all of these nodes were running on the same EC2 host and concluded that the instance was suffering from disk IOPS saturation. Forty-five minutes into the attack, however, nearly all agents are again caught up to the head of the blockchain.

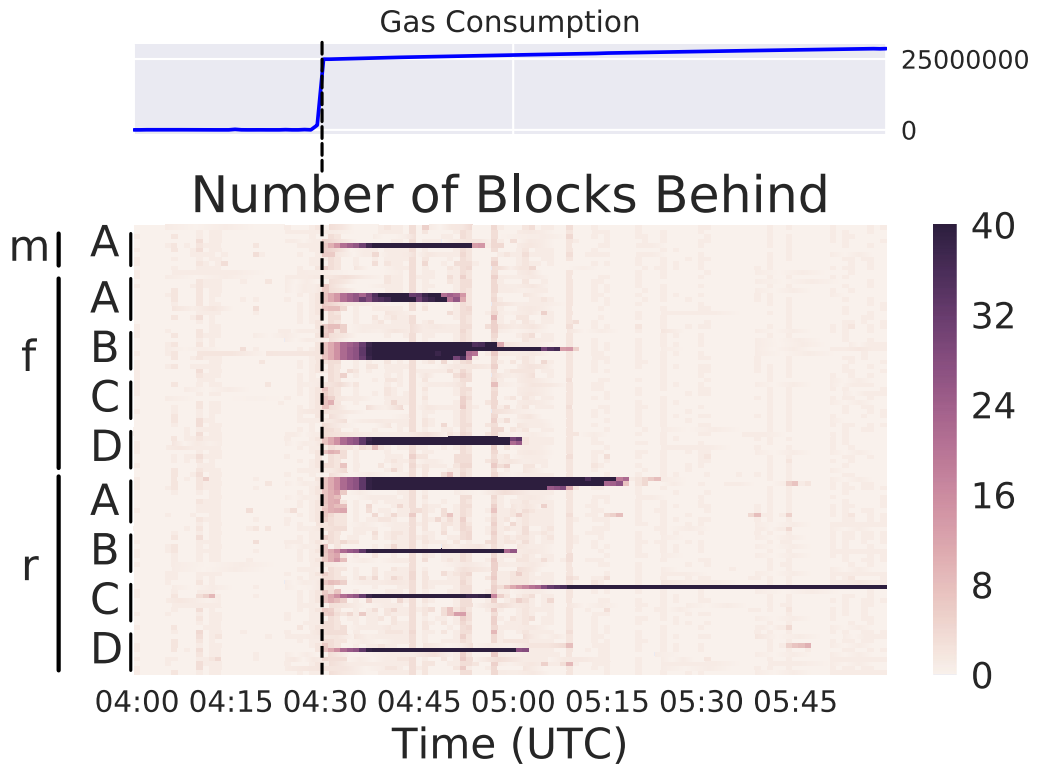


Fig. 12. Number of blocks behind the head of the chain over time (x) for each node. The y-axis breaks nodes down by role: miner (m), agent with a full set of 20 peers (f), and agent restricted to two peers (r) as well as by the net classes defined in Table 2.

Remote Agent. For embedded systems unable to participate reliably in the blockchain directly, a solution 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.

6.2 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 1ms and 1.5ms, 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

Table 5. Summary of the Resources Associated with HPL Microservices in Three Deployments

Deployment	Length	Device Type	# Resources
Campus Building	13 Months	Thermostats	21
		WSN Mote	64
		Power Meter	5
Residential House	17 Months	Thermostats	84
		WSN Mote	300
		Power Meter	12
Research Lab	13 Months	WSN Mote	1,485
Air Velocity	11 Months	Sensor	12

cryptography is not onerous. On the 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.

6.3 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 [16]) or consuming its sensor feeds (e.g., Mortar.io [26]). This makes it difficult to configure layer-3 firewalls to block illegitimate traffic upstream. In BOSSWAVE, the device and its agent only speak 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.

6.4 Deployments

Aside from the city-scale emulation, we also present details about four deployments of BOSSWAVE and XBOS, as summarized in Table 5. 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 5 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.

7 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 [17], Sensor Andrew [29], Mortar.io [26], and the like. 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 [6], 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 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 Refs [17, 26], 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 [12] and leveraging the extensive libraries created for this purpose [9, 10]. 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.

8 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

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

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

We gratefully acknowledge the assistance of 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 4th 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 *3rd International Workshop on Agent Technologies for Energy Systems, Valencia, Spain*.

[3] Michael Andersen, John Kolb, Kaifei Chen, David E. Culler, and Randy Katz. 2017. Democratizing authority in the built environment. In *Proceedings of the ACM International Conference on Embedded Systems for Energy-Efficient Built Environments (BuildSys). ACM*.

[4] Michael P. Andersen, John Kolb, Kaifei Chen, Gabriel Fierro, David E. Culler, and Raluca Ada Popa. 2017. WAVE: A decentralised authorization system for IoT via blockchain smart contracts. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2017/EECS-2017-234.html.

[5] 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.

[6] 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 4th ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*. ACM, 80–87.

[7] 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*.

[8] David Belson. 2016. Akamai state of the internet connectivity report, Q4 2016. (Nov. 2016).

[9] UC Berkeley. 2017. BOSSWAVE Golang library. Retrieved from https://github.com/immesys/bw2bind.

[10] UC Berkeley. 2017. BOSSWAVE Python library. Retrieved from https://github.com/SoftwareDefinedBuildings/bw2python.

[11] UC Berkeley. 2017. BOSSWAVE source code. Retrieved from https://github.com/immesys/bw2.

[12] UC Berkeley. 2017. XBOS documentation. Retrieved from https://docs.xbos.io/.

[13] 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. DOI: https://doi.org/10.1109/RTSS.2013.36

[14] Kaifei Chen, Jonathan Fürst, John Kolb, Hyung-Sin Kim, Xin Jin, David E. Culler, and Randy H. Katz. 2017. SnapLink: Fast and accurate vision-based appliance control in large commercial buildings. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 1, 4 (2017), 129:1–129:27.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] Alan A. A. Donovan and Brian W. Kernighan. 2015. *The Go Programming Language*. Addison-Wesley Professional.

[20] 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.
[21] The Linux Foundation. 2017. Kubernetes. Retrieved from https://kubernetes.io.
[22] 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 *Proceedings of the 2012 IEEE PES Innovative Smart Grid Technologies (ISGT)*. IEEE.
[23] Tridium Inc. 2017. Niagara 4. Retrieved from https://www.tridium.com/products-services/niagara4.
[24] John Kolb. 2018. *Spawnpoint: Secure Deployment of Distributed, Managed Containers*. Master's thesis. EECS Department, University of California, Berkeley. Retrieved from http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-1.html.
[25] Andrew Krioukov, Gabe Fierro, Nikita Kitaev, and David Culler. 2012. Building application stack (BAS). In *Proceedings of the 4th ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*. ACM, 72–79.
[26] 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.
[27] 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 *Proceedings of the 2015 IEEE Innovative Smart Grid Technologies-Asia (ISGT ASIA)*. IEEE, 1–6.
[28] David R. Raymond and Scott F. Midkiff. 2008. Denial-of-service in wireless sensor networks: Attacks and defenses. *IEEE Pervasive Computing* 7, 1 (2008), 74–81.
[29] Anthony Rowe, Mario E. Berges, Gaurav Bhatia, Ethan Goldman, Ragunathan Rajkumar, James H. Garrett, José M. F. 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.
[30] Chenguang Shen, Rayman Preet Singh, Amar Phanishayee, Aman Kansal, and Ratul Mahajan. 2016. Beam: Ending monolithic applications for connected devices. In *Proceedings of the 2016 USENIX Annual Technical Conference (USENIX ATC'16)*. USENIX Association, 143–157.
[31] Péter Szilágyi. 2015. eth/Fast Synchronization Algorithm. Retrieved from https://github.com/ethereum/go-ethereum/pull/1889.
[32] Jay Taneja, Andrew Krioukov, Stephen Dawson-Haggerty, and David Culler. 2013. Enabling advanced environmental conditioning with a building application stack. In *Proceedings of the 2013 International Green Computing Conference (IGCC)*. IEEE, 1–10.
[33] Gavin Wood. 2014. Ethereum: A secure decentralised generalised transaction ledger. *Ethereum Project Yellow Paper* 151 (2014), 1–32.
[34] XMPP Standards Foundation. 2017. XMPP. *R*etrieved from https://xmpp.org.

Received January 2018; accepted March 2018