Playground: A Safe Building Operating System

Xiaohan Fu*, Yihao Liu[†], Jason Koh[‡], Dezhi Hong[§], Rajesh Gupta*, and Gabe Fierro[¶]
*University of California San Diego, USA, {xhfu, rgupta}@ucsd.edu
[†]Nanyang Technology University, Singapore, yihao002@e.ntu.edu.sg
[‡]Mapped, USA, jason@mapped.com
[§]Amazon, USA, hondezhi@amazon.com
[¶]Colorado School of Mines, USA, gtfierro@mines.edu

***Abstract*—Building operating systems are an emerging class of system software that provides services to applications running on commercial buildings. The current state-of-the-art requires applications to be trusted and carefully monitored due to a lack of authorization, access control, and execution isolation mechanisms in existing building operating systems. Proposed solutions do not adequately handle the complexity and scale of modern buildings, therefore impeding the adoption of applications that can enhance energy efficiency, occupant health, comfort, and productivity.**

This work explores the execution of *untrusted* user-facing applications in smart building environments with a focus on maintenance and management labor costs, ensuring the practicality and long-term sustainability of adopting such applications. We develop an operating system abstraction for smart buildings, PlayGround, that incorporates a structured semantic representation of the building to inform the safe, multi-tenant execution of untrusted applications. We use the semantic representation to implement (a) a novel graph-based capability mechanism for fine-grained and expressive access control management, and (b) a resource isolation mechanism with preemptive interventions and passive telemetry-based live resource monitoring. We demonstrate PlayGround on several real applications in a real building.

***Index Terms*—Brick, building, isolation, capability**

I. Introduction

Buildings are at the center of nearly all human activities. An average American spends 90% of time in a building [1]. A 2022 survey found that buildings account for 40% of the total energy consumption in the U.S and 20% of global energy consumption, with the latter number rising 2% per year [2]. Consequently, there is a need to reduce building energy consumption while also maintaining occupant comfort and increasing productivity. The rise of the Internet of Things and the subsequent emergence of buildings as connected digital assets offers new opportunities to achieve these goals through the adoption of intelligent, data-driven "applications."

Applications need to run somewhere. Existing building management systems (BMS) implement supervisory control and basic alarming, but their proprietary nature makes them difficult to program. Over a decade of work on operating systems for buildings [3]–[6], application runtimes [7]–[9], and applications [10]–[13] has established the need for programmatic abstractions over complex building subsystems and the I/O functionality provided by the underlying BMS. Prior work has also proven the difficulty of doing this effectively across a vastly heterogeneous building stock. Every building is a "one-off": an ad-hoc collection of equipment and software from multiple vendors, most of which are non-interoperable. The resulting need for standardized representations of buildings has inspired academic, federal, and commercial investment in semantic metadata ontologies and schemas like Brick [14] and others [15]–[18]. These semantic digital representations increase the programmability of buildings by allowing applications to dynamically discover available resources and subsystems.

Despite technological and standardization victories, the pace of innovation and the adoption rate of "smart building applications" remains low for two safety reasons. First, building managers (the custodians of digital access to buildings) are naturally protective of the safety- and comfort-critical elements of buildings and are reluctant to permit unvetted and possibly buggy applications to influence the building's operation. Second, the inability of modern buildings to properly restrict an application's permissions [5], [19] can make the application's execution opaque to the manager, leaving it unclear what the application is actually doing or how it is affecting the building's operation. Indeed, each of the applications cited above was either evaluated in simulation or with the carefully curated trust and watchful eye of a building manager. We therefore propose two safety properties that a building application runtime should provide: 1. (principle of least privilege) ensure that applications interacting with digital and physical building resources only access what is necessary, 2. (resource isolation) enforce that the impact of applications on building resources are constrained within bounds set by the building managers, including potentially indirectly affected ones such as temperature, energy, and peak power due to the interconnected nature of building systems.

In this paper, we propose PlayGround, a "safe" operating system (OS) abstraction for buildings that enables the execution of *untrusted, multi-tenant* applications in modern buildings. The goal of PlayGround is to encourage innova-

This work is partly supported by the National Science Foundation under Award Number 1947050.

tion and exploration of how modern building applications can provide value to occupants, managers, and other stakeholders while avoiding the intensive manual effort required to deploy them *safely*. While prior work has proposed various access control [5], [20] and execution isolation [3], [21] mechanisms, these require intricate configuration and can impose substantial maintenance burdens on building managers. We integrate a semantic representation of the building — Brick [14] — to imbue our OS services with detailed knowledge of the building, allowing it to perform automatically the same procedures and checks that would normally be performed by the building manager, and to a greater degree of fidelity. This also allows all system configurations to be specified declaratively.

Specifically, our contributions are: **1**) a detailed description of a safe building OS capable of running untrusted user applications, comprising: **2**) a dynamic fine-grained semantic access control mechanism, and **3**) a resource isolation mechanism that is aware of the topology and composition of building systems.

II. Background and Prior Work

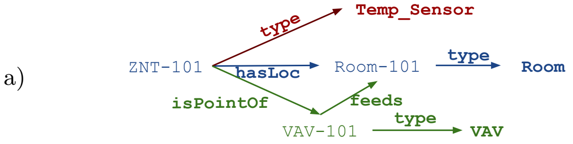
A. BMS and Building Control Logic

Building Management Systems (BMS) monitor and control various building parameters by controlling subsystems like heating, ventilation, and air-conditioning (HVAC) systems, lighting systems, and fire safety systems. Among them, HVAC systems have gathered much research interest for their significant energy impact. Air Handling Units (AHUs), which handle the conditioning of the air supplied to the building, and Variable Air Volume (VAV) boxes, which recondition the air supplied to each zone, are common control objectives in HVAC systems.

Crucially, although BMS are typically isolated from the public internet behind a firewall or airgapped network, there is often little security inside the building network. Some building network protocols do provide support for modern access control mechanisms [22], but widely deployed protocols like BACnet provide neither authorization nor authentication mechanisms within the network. Support for auditing writes within the BMS is also rare. As a result, building managers must be careful to admit only trusted and carefully tested applications into the network.

B. Computing and Building Operating Systems

Our proposed approach to a safe building operating system takes inspiration from classical operating systems (OS). Classical OS provide safe abstractions of underlying hardware that allow many untrusted applications to execute over shared resources, where a centralized trusted kernel operates in a protected memory region to provide several essential services. These include a hardware abstraction that insulates users and applications from the complexities of direct hardware interaction and allows the OS to multiplex access, a scheduler that synchronizes the execution of applications and handles the assignment of OS resources, strict memory isolation between processes to ensure misbehaving processes do not interfere with the operation of the system and other processes, and finally, access control mechanisms that ensure that users and applications do not gain unauthorized access to resources. These abstractions are key to providing *multi-tenancy*, the ability of an OS to support the safe execution of multiple independent applications over shared resources.



b)

```
prefix brick: <https://brickschema.org/schema/1.3/Brick#>
prefix ex: <http://example.com/ex>
select ?s where {
   ?vav brick:feeds ex:Room-101 .
   ?vav a brick:VAV . # "a" means is type of
   ?vav brick:hasPoint ?s .
   ?s a brick:Temperature_Sensor . }
```

Fig. 1: a) An example Brick graph describing the composition of a VAV, a room, and a temperature sensor (prefixes ex: omitted for brevity). b) An example SPARQL query that finds the temperature sensor(s) of the VAV(s) that feeds ex:Room-101.

Prior work proposes that operating systems for buildings should provide many of the same properties as traditional OS [3]. Indeed, there are several similarities: buildings contain heterogeneous hardware that must be abstracted away to simplify application development; buildings must also maintain safety- and comfort-critical constraints like indoor temperature and air quality. Other academic efforts augment the building OS with additional services, like data archival [3], [4], rule-based data sharing [21], and decentralized authorization [5], [20].

C. Building Semantic Metadata

Semantic metadata is an emerging family of formal digital representations of buildings that encode cross-subsystem, vendor-agnostic, and machine-readable descriptions of the resources, assets, and I/O points in buildings [23]. They replace existing ad-hoc and unstructured labeling schemes that underlie existing approaches to categorizing building data [14], [24]. The lack of standard representations of buildings and data is a major factor impeding the adoption of intelligent building applications [25].

Efforts to standardize digital descriptions of buildings have crystallized around graph-based models built with the Resource Description Framework (RDF) W3C standard. These include Brick [14], RealEstateCore [17], Building Topology Ontology (BOT) [26], and ASHRAE Standard 223P [16]. Building platforms and systems have been proactively integrating them across industry and academia sectors [7], [8], [18]. Extensive research has also been done on efficiently generating the metadata description of a building based on existing information from BMSes and others [24], [27]–[30].

Our system incorporates the Brick metadata schema. A Brick model is a graph that represents the assets and

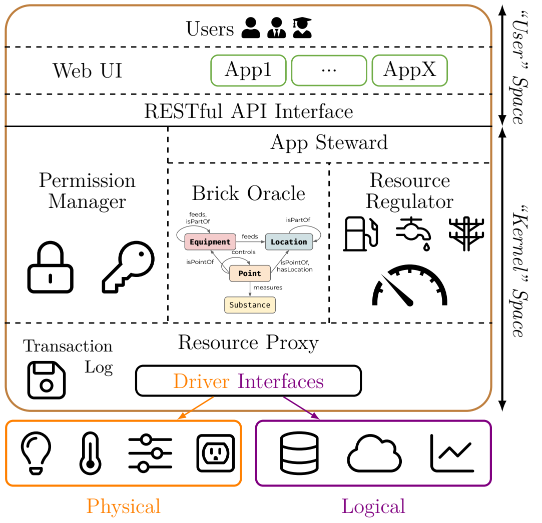


Fig. 2: Architecture of PlayGround with a conceptual "kernel"/"user" space boundary and five kernel components.

I/O points in a building, and the relationships between them. This captures the composition and topology of the building formally, permitting the OS to reason about how application actions can influence each other, the building, and the occupants. A Brick model (graph) is expressed as a set of *subject-predicate-object* triplets, each encoding a relationship (*predicate*) from a *subject* entity to an *object* entity. Entities and relationships are identified by URIs, but are typically written in an abbreviated form (e.g. `brick:Temperature_Sensor`). An example Brick graph is shown in Figure 1.a. Applications query Brick graphs using the standard SPARQL query language: the query in Figure 1.b will return `ex:ZNT-101` on the example graph.

III. System Design

PlayGround features a conceptual separation of user/kernel space analogous to classical OS. Our kernel design has five major components as illustrated in Figure 2: 1) Resource Proxy, 2) Brick Oracle, 3) Permission Manager, 4) Resource Regulator, 5) App Steward. The Resource Proxy provides the hardware abstraction and handles actual access to various cyber-physical building resources such as sensors, actuators, data storage, and alarms. The Brick Oracle contains a queryable Brick model representing a semantic, graph-based representation of the building's structure, subsystems, and I/O points. The Permission Manager is an authorization service that manages and enforces the access control (capability) policies for various entities in the system through a novel graph-based mechanism. The Resource Regulator enforces two forms of resource isolation: 1) per-action value guards inspecting application writes, and 2) live resource tracing mechanisms tracking consumption constantly. Finally, the App Steward manages the life cycle of each app instance from registration to termination.

Applications are separated from the kernel services through a RESTful API service provided by the system. Some service endpoints are privileged (*e.g.,* configurations on these kernel components) and can only be performed by privileged users *i.e.,* building managers; all others require authorization by the Permission Manager. We then elaborate on the design of each component below.

A. Brick Oracle

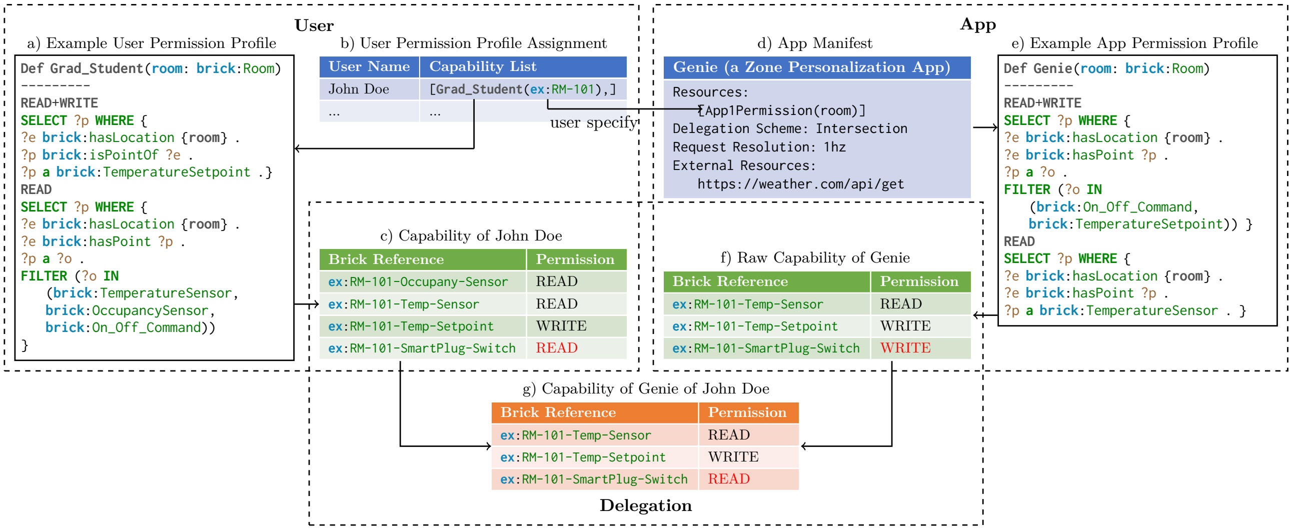
The Brick Oracle is a graph database engine hosting the Brick models of any buildings under the supervision of a PlayGround instance. A building's Brick model stores rich semantic data on the building's architecture, the assets and equipment within the building, the topology and structure of the building's subsystems (*e.g.,* HVAC, lighting), the data sources and command points presented by the building's BMS, and how all of these elements relate to one another. The graph database exposes a SPARQL query endpoint which allows applications to 1) retrieve metadata about buildings and building resources, and 2) define (sub)sets of resources. This feature is enough to support existing metadata-based application runtimes and frameworks [7], [8], [23].

The choice of Brick as the metadata representation building is significant. Brick's design enforces uniformity in how buildings are modeled; this is further assisted by new automated workflows [25], [27], [31]. Normalizing how buildings are modeled makes it possible for building managers, through PlayGround, to define access control policies, identify resources, and configure applications through queries against a building's model. The configuration of PlayGround is largely defined *declaratively* by queries against the Brick Oracle (§III-D,§III-E).

This has two advantages. First, this reduces the effort required to manage complex access control policies. Rather than managing long flat lists of equipment names and points, PlayGround allows policies to be defined by how resources relate not just to one another, but also to the building's occupants, application users, and the building itself. Second, as the underlying building evolves through natural churn (repairs, remodels, etc), policies can be automatically refreshed so they remain accurate.

B. Resource Proxy

The Resource Proxy provides read or write access to physical and logical I/O "points". This includes physical points such as the building's sensors and actuators, and logical points like timeseries histories and alarms. The Resource Proxy abstracts away the different protocols and APIs required to access these resources, and unifies access behind a generic read/write interface. Each point is represented in the Brick model; thus, the read/write interface just requires a Brick reference (the name of the point in the Brick model) to direct the payload to the correct location. Writes take a value and a set of standard flags; reads take a time range. The Brick model captures whether a point supports read-only, or read-write operations. The Resource Proxy logs all read/write



a) Example User Permission Profile

```
Def Grad_Student(room: brick:Room)
---------
READ+WRITE
SELECT ?p WHERE {
?e brick:hasLocation {room} .
?p brick:isPointOf ?e .
?p a brick:TemperatureSetpoint .}
READ
SELECT ?p WHERE {
?e brick:hasLocation {room} .
?e brick:hasPoint ?p .
?p a ?o .
FILTER (?o IN
   (brick:TemperatureSensor,
   brick:OccupancySensor,
   brick:On_Off_Command))
}
```

b) User Permission Profile Assignment

User Name	Capability List
John Doe	[Grad_Student(ex:RM-101),]
...	...

c) Capability of John Doe

Brick Reference	Permission
ex:RM-101-Occupany-Sensor	READ
ex:RM-101-Temp-Sensor	READ
ex:RM-101-Temp-Setpoint	WRITE
ex:RM-101-SmartPlug-Switch	READ

d) App Manifest

Genie (a Zone Personalization App)
Resources: [App1Permission(room)] Delegation Scheme: Intersection Request Resolution: 1hz External Resources: https://weather.com/api/get

e) Example App Permission Profile

```
Def Genie(room: brick:Room)
---------
READ+WRITE
SELECT ?p WHERE {
?e brick:hasLocation {room} .
?e brick:hasPoint ?p .
?p a ?o .
FILTER (?o IN
   (brick:On_Off_Command,
   brick:TemperatureSetpoint)) }
READ
SELECT ?p WHERE {
?e brick:hasLocation {room} .
?e brick:hasPoint ?p .
?p a brick:TemperatureSensor . }
```

f) Raw Capability of Genie

Brick Reference	Permission
ex:RM-101-Temp-Sensor	READ
ex:RM-101-Temp-Setpoint	WRITE
ex:RM-101-SmartPlug-Switch	WRITE

g) Capability of Genie of John Doe

Brick Reference	Permission
ex:RM-101-Temp-Sensor	READ
ex:RM-101-Temp-Setpoint	WRITE
ex:RM-101-SmartPlug-Switch	READ

Fig. 3: A sample derivation procedure of the capability of the app *Genie* on behalf of the user *John Doe*. Note the effect of the *intersection* delegation scheme (marked in red). For brevity, we list "WRITE" instead of "READ+WRITE" in subfigures c, f, and g.

requests in a transaction log with additional metadata including the timestamp and identity of the requester. Other components in the system and building managers may use this information to relinquish controls, investigate users/apps, and so forth. Due to the simplicity of the read/write interface, it is also straightforward to extend the Resource Proxy to new types of building resources.

C. App Steward

The App Steward manages the entire life cycle of each application including registration, execution, and termination. PlayGround hosts every application locally by loading their front-end as external resources into the web UI of PlayGround and executing their backend servers in isolated containers. By default, all app containers can only access the system's API. This way PlayGround ensures full control over the applications and their data flow.

1) Registration: Every application needs to be approved by the building managers to be "installed" into the system on registration. To be able to host them, we require app developers to upload the code of both the frontend interface and the backend service per registration. We also require the developers to provide a manifest specifying the permission profile, action frequency, and external endpoints of the application. The permission profile and delegation scheme together define the capability required by this app in a graph-based format (§III-D). The action frequency specifies the maximum rate that the app can make requests at. The external endpoints are vital resources other than the system API endpoints that this application would need to function properly.

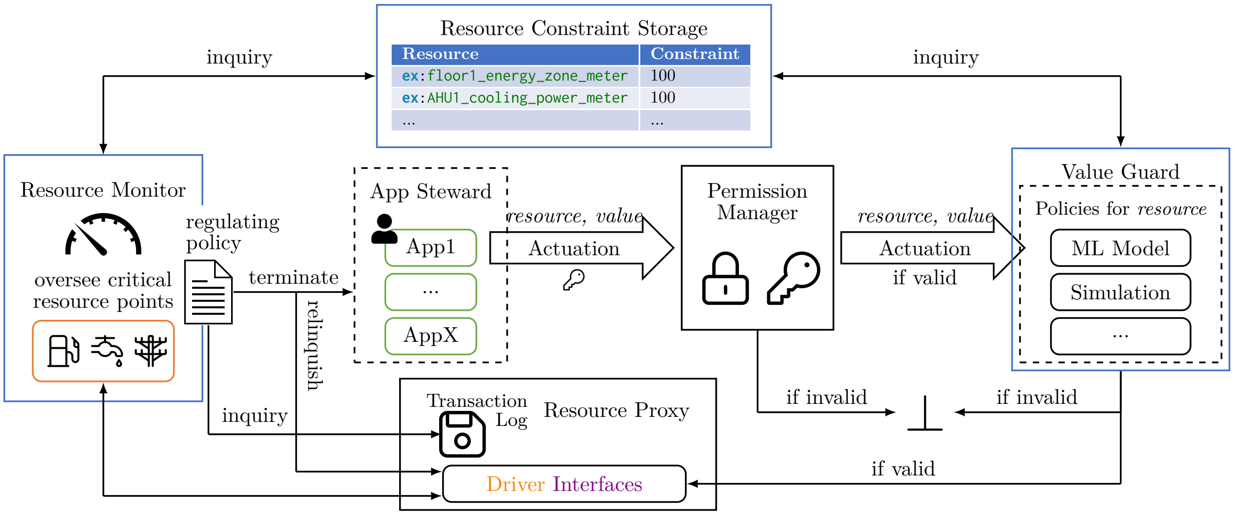
Figure 3.d contains an example app manifest. This example application wants to make requests at most every second and access an external weather service API endpoint. Building managers may inspect the app manifest (and optionally the code) to decide whether or not to approve it, before it can be instantiated by users. Approvals can be reverted, resulting in the termination of existing instances and the removal of the app from the system.

2) Execution: Each application instance for each user runs in its own container. This allows PlayGround to precisely monitor the frequency and content of requests against the building, as well as ensure the application is only accessing resources the associated user has permission for. Compared to prior work where one external application serves all users on the system, this effectively prevents over-privileging applications.

3) Termination: An app instance can exit normally, be gracefully terminated, or be forcefully killed. An app will exit normally if a user is done with using it by clicking the exit button or being idle for too long. Sometimes, the system may require an app instance to be stopped *e.g.,* the app is disapproved or this instance is breaking resource constraints (§III-E). We offer a mechanism to allow the app to terminate gracefully after receiving the request *e.g.,* cleaning up things and relinquishing certain controls, but the App Steward may still force kill an app instance (and relinquish all relevant points) when necessary.

D. Permission Manager

Permission Manager is the authorization service in our system; it also plays a key role in resource isolation. The capability of each entity (user/app) in our system is managed in a novel graph-based format we call permission profile. A permission profile is a function-like object that takes in a few Brick-typed arguments and returns two lists of building resources for read and (read plus) write capability respectively. The derivation of the two capability lists is defined by two SPARQL queries which use the provided arguments as parameters. Note that the permission profile is statically typed — arguments must be of the specified Brick class. Upon any read/write access to building resources, the Permission Manager checks the



Resource	Constraint
ex:floor1_energy_zone_meter	100
ex:AHU1_cooling_power_meter	100
...	...

Fig. 4: The system components involved in a user/app making an actuation (write) request to a resource point.

capability of the requester to verify whether this is an authorized attempt and approves/declines accordingly.

1) User Permission: To define a user's capability, the building manager just needs to choose or create a proper permission profile and specify the argument(s). Multiple permission profiles can be applied to one user — the actual capability of this user is defined by the union of these profiles. Consider Figure 3: User John Doe's capability is defined by one single permission profile named **`Grad_Student`** with argument `ex:RM-101`. The SPARQL query results define the points this user is authorized to read/write. If there is a need to update the capability of John Doe, the building manager can simply change the arguments passed to **`Grad_Student`** or assign more permission profiles to this user. The **`Grad_Student`** permission profile can be reused on other users with potentially different parameters. The building manager may also directly edit the SPARQL queries in the profile to enforce updates on the capabilities of all users under this permission profile at once.

2) App Permission: Recall that applications are required to provide a manifest listing the permission profile and the delegation scheme per registration. Users instantiate applications with arguments to the application's permission profile. The arguments must be from the user's capability list or the arguments used to instantiate any of the user's permission profiles. This ensures the application cannot escalate privileges beyond what the user can do. For the example in Figure 3, John Doe can instantiate Genie with the four entities in Figure 3.c and `ex:RM-101` (the argument of his permission profile **`Grad_Student`**), out of which `ex:RM-101` is the only option of required type `brick:Room`. The capability of this app instance, as listed in Figure 3.f, is then computed similarly by the SPARQL queries defined in permission profile **`App1Permission`** (Figure 3.e) with argument **`Grad_Student`**.

When an app instance is making write or read requests to resources, its capabilities are *delegated* by the user. PlayGround provides two types of delegation schemes: intersection and augmentation. App developers specify the desired delegation scheme in the manifest and let the building manager decide whether it's appropriate. The intersection scheme is intuitive — the final capability of an app on behalf of a user is the intersection of the capability of this app instance and the user. In our example, though genie app should have read plus write capability to `ex:RM-101-SmartPlug-Switch`, it is degraded to read after the intersection since John Doe has only read capability to it (highlighted in red in Figure 3). Augmentation grants the app instance the full capability as defined by its permission profile, regardless of whether the user has those capabilities or not. This is particularly useful when the application may need to access low-level resources that are not authorized for regular users. For example, to fulfill a user's zone comfort request, an HVAC control app may need to operate on AHU setpoints that are inaccessible to the user.

Permission profiles provide a fine-grained and adaptive definition of capabilities and effectively avoid over-privileged scenarios due to gross-grained policies over fixed domains like buildings or floors. Meanwhile, the maintenance and extension of them are as expressive and simple as writing/modifying Brick queries and arguments.

E. Resource Regulator

The Resource Regulator implements resource *isolation*. We allow building managers to specify constraints on the value of critical building resources *e.g.,* peak power consumption, and to customize regulation policy *i.e.,* actions to take once the constraints are broken. This allows building managers to bound the behavior of untrusted applications to avoid them "crashing" the building. Resource Regulator has three main sub-components, described below.

1) Resource Constraint Storage: The table of resource constraints relates Brick references (representing I/O points and building resources) to their constraints. For example, in Figure 4 we set an upper bound of 100 on ex:AHU1_cooling_power_meter and 100 on ex:floor1_energy_zone_meter. The first constraint refers to a physical energy meter; the second refers to a virtual meter that aggregates energy consumption (defined declaratively by a query against the Brick Oracle). The Resource Regulator ensures that the value of the resources does not exceed the corresponding constraints. Constraints apply not only to read-only resources such as sensor readings but also to writable resources such as actuators and setpoints.

2) Pre-write Value Guard: Pre-write value guard is a preemptive mechanism to ensure resource isolation. It looks at the proposed write value of an authorized write request and only approves it if the value is decided to be valid. The decision on validity is made by the validators assigned to the resource targeted by the write request. Value validators are functions that take the target resource and the write value as inputs and return a boolean value indicating the validity or otherwise raise an error on failure to make the decision. Building managers can assign a value validator to a set of resources defined by a SPARQL query — this assignment is adaptive and fine-grained. Building managers may also assign more than one validator to a set of resources to form a queue of validators. The value guard will only approve the request when all validators in the queue approve (failed ones skipped). The system defaults to refusing actions against resources without a validator. For the example in Figure 4, we assign various example validators including "ML Model" and "Simulation" to the target entity. If the "ML Model" fails and the "Simulation" validator returns false, value guard will immediately disapprove this request, skipping any subsequent validators. Validators can always query the resource constraint storage and Brick Oracle for necessary information.

3) Live resource monitor: Live resource monitor is a passive isolation mechanism. The resource monitor continuously watches critical resource points as set up by the building managers (*e.g.,* resource points that measure or represent power or energy consumption) and takes actions following the provided regulating policy when the values of the watched resources exceed the constraints specified in resource constraint storage (see Figure 4). The action and the subject of the action are fully up to the policy — it may freely interact with other kernel components such as the Resource Proxy, Brick Oracle, and App Steward to obtain useful information and carry out the action. For example, the default fallback policy in our system will relinquish all relevant control points and terminate apps that have written to those points.

IV. Implementation

We have built a prototype of PlayGround to validate our design ideas. Here we describe our implementation as a reference (see Appendix for more details). PlayGround comprises a set of Docker containers. One container implements the functionalities of each kernel component of our design described in §III and offers a RESTful API interface exposed to the users. The other containers host database services to store essential information in the kernel, including an RDF-graph database for Brick modelings and SPARQL queries, an object database for system policies such as application manifests and permission profiles, and a time-series database for transaction logs. Privileged users, *i.e.,* building managers, may manage the system policies and non-privileged users may initiate read/write requests and Brick queries through the API endpoints or a basic web UI. Users instantiate and interact with installed apps through the web UI.

The declarative aspects of our design rely on SPARQL queries. To speed up these queries, we implement in-memory caching and pre-computation. When the system starts, a background thread pre-computes frequently used queries such as those for capability derivation and resource-validator mappings. These pre-computed results are then stored in the cache in the format of key-result pairs, where key is the identifier of the query plus query inputs. Uncached queries are computed on-demand and cached. PlayGround suspends all system services of a building while its Brick graph is updated. After the graph of a building has been updated, all related cache entries are invalidated and the pre-computation process restarts. We evaluate the effect of caching in §V-E.

V. Evaluation

To evaluate the efficacy of our system, we first inspect our two proposed mechanisms for access control (Permission Manager) and resource isolation (Resource Regulator) with two real-world case studies respectively. We then conduct the microbenchmark showing that the runtime overhead of these mechanisms is reasonable and scales well. We conclude with real-world deployment observations.

A. Setup

For the following two case studies, we deploy PlayGround on a real-world multi-purpose building on campus. We support the control of the HVAC system and 80 smart plugs through a BACnet driver interface in the Resource Proxy. In companion, we have a Brick representation of this building which describes arrangements of rooms and floors, and specifications of hardware including over a thousand HVAC points and the smart plugs (hardware limit, external reference to the BACnet objects, etc.) with their relationships to rooms/floors. The building prefix is again ex:. We conduct the case study on one AHU (ex:AHU1) and two offices (ex:RM-101, ex:RM-102) fed by this AHU to minimize the affected residents.

B. Case Study #1 on Access Control

In this first case study, we evaluate our graph-based capability (access control) system with the zone personalization application, Genie [13]. Genie allows users to personalize the HVAC settings in their zone by adjusting the temperature setpoint. Users can also control

TABLE I: Detailed timeline of the case study #1 on access control.

#	Event and Effect
1	Two new grad student users A and B signed up to the system. BM assigned `Grad_Student` permission profile to them with arguments `ex:RM-101` and `ex:RM-102` respectively. *Our system derives the capability of new users A and B.*
2	A and B instantiated Genie with `ex:RM-101` and `ex:RM-102` as the argument of Genie's permission profile respectively. *The actual capability of the two genie app instances delegating A&B were derived (refer Figure 3 for details).*
3	A and B both adjusted the temperature setpoint for their respective rooms successfully. A tried to control the smart plug but got "permission denied". `Grad_Student` *profile allows only read access to smart plugs.*
4	A reported this. BM then updated the `Grad_Student` profile by rewriting its R/W query to include the smart plug (`brick:On_Off_Command`) in the results (see updated one in Appendix E). *Capabilities of A and B were updated noting the updated profile* `Grad_Student`. *The capability of two app instances on behalf of them was also re-derived subsequently by the system.*
5	A retried to control the smart plug and was successful. *The updated capability took effect instantly.*
6	BM moved the smart plug from RM-101 to RM-102 and updated the Brick graph of the building accordingly. *Our system recomputed the above capabilities noting the changes in the building environment.*
7	A and B both adjusted the temperature setpoint for their respective rooms successfully. A tried to control the smart plug but got "resource not found". B successfully controlled the plug. *Our system denied A's request but approved B's request since the plug was no longer in RM-101 but in RM-102.*

the smart plug within the zone with it. We ported Genie to use the interface provided by our system kernel — given a provided room, it finds the temperature setpoint and the smart plugs in the room through a Brick query and controls them through Resource Proxy.

We start with the initial state of the system of this case study. Recall that the capability of a user or app is defined by permission profiles composed of Brick queries and the provided arguments. The building manager (BM) reused the example permission profile `Grad_Student` we've shown in Figure 3 and incorporated it into the system. (BM can always write more Brick queries to define additional permission profiles. See Appendix E.) Meanwhile, BM approved the app Genie, with the app manifest and permission profile we've seen in Figure 3. BM placed a smart plug in RM-101, represented as a `brick:On_Off_Command` entity as a point of `ex:RM-101` in the Brick graph.

We list actions taken by users during the case study and the effect on our system in Table I. We also plot the trace of relevant hardware readings in Figure 5 along with the event timeline. We can see that temperature sensor readings were changing according to user actions on the temperature setpoint (with slight ventilation delay) and smart plugs were turned on and off when the control actions were successful (at the time of event 5 and 7).

Observations: Event 1, 4, and 5 show the flexibility of our capability system. Building managers can easily create and assign capability policies, and they can also update existing policies without worrying about running applications. Event 2, 3, 6, and 7 demonstrate that our graph-based permission profile and per-user per-app app instance design ensures access control at fine granularity avoiding over-privileged cases — app instances of the same app (genie) delegating user A and B were defined with exclusive capabilities and couldn't interfere with each other's resources. Event 6 and 7 further exhibit that our (capability) system may work adaptively throughout building environment changes — after moving the smart plug, Genie instance of A could no longer find the plug as expected and meanwhile the instance of B gained the capability to control it.

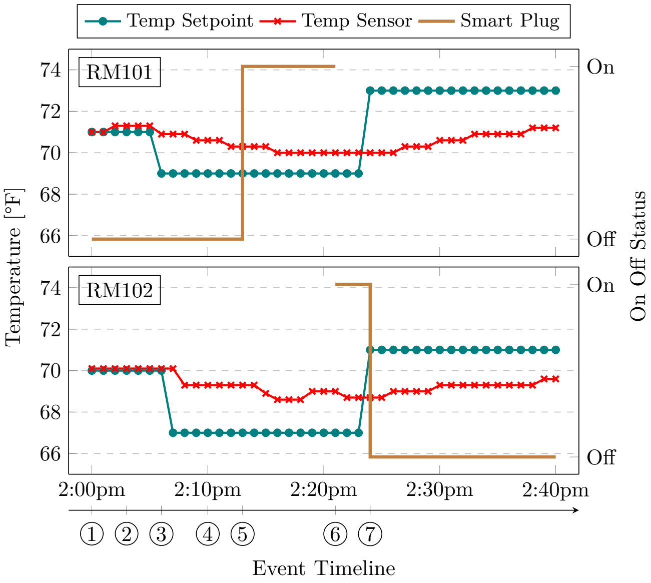


Fig. 5: Zone thermal conditions in Case Study #1. Circled numbers refer to the event number in Table I.

C. Case Study #2 on Resource Isolation

In this second case study, we evaluate the efficacy of our Resource Regulator in maintaining resource isolation constraints and preventing authorized yet potentially harmful actions from (untrusted) applications. We prepare a sample application that can control the points of a given AHU. The app comes with a permission profile that uses the augmentation scheme and asks write access to all points of the given argument (an AHU). This ensures that every action taken by an instance of this app on the assigned AHU in its argument is authorized by our capability system. The cooling command of the AHU, which decides how much cooling (%) the AHU will produce, will be the main test subject of this case study.

At the initial state of the case study, BM approved and installed this app. We also set up a test user with write capability to `ex:AHU1` and its points. Recall that the resource constraint policies in Resource Regulator are managed by resource constraint storage, resource monitor, and pre-write value guard. To protect hardware and

TABLE II: Detailed timeline of case study #2 on resource isolation.

#	Event and Effect
1	Wrote 0% to the cooling command, successful.
2	Wrote -100% and 200% to the cooling command, both failed. *Blocked by the range checker.*
3	Wrote 95% to the cooling command, unsuccessful. *Blocked by the power predictor since the prediction is 103 kBTU/h, larger than the 100 kBTU/h constraint set.*
4	BM updated the brick graph such that now the maximal of the cooling command is 80%.
5	Wrote 95% to the cooling command, unsuccessful. *Blocked by the range checker instead, for its higher priority.*
6	Wrote 75% to the cooling command, successful.
7	BM updated the upper bound of `ex:AHU1_cooling_power_meter` from 100 to 60 kBTU/h in the constraint storage.
8	System relinquished the cooling command and killed this app. *The resource monitor spotted the cooling power had exceeded the constraint and took action following the regulating policy.*

save energy, BM set an upper bound on the cooling power meter of AHU1 (`ex:AHU1_cooling_power_meter`) at 100 kBTU/h in the constraint storage and let the resource monitor oversee it. The regulating policy is configured to relinquish the cooling command and terminate any existing app instances that have written to it when the cooling power is over the above constraint. BM also set up two validators on the cooling command of `ex:AHU1` in the value guard (as ordered in the queue): 1) a *range checker* that checks whether the input cooling command value falls between the maximal and minimal defined in Brick graph (0 and 100 in this case); 2) a *power predictor* that predicts the cooling power of an AHU given the input cooling command based on historic data using a naive two-layer neural network and checks whether it is smaller than the upper bound defined in the resource constraint storage.

We describe the detailed timeline of actions taken by users in this case study in Table II and plot the cooling command/power of AHU1 throughout this case study in Figure 6. We can see that the cooling power and cooling command were changing according to user actions at event 1 and 5. Also, note that the cooling command was relinquished and dropped to 0% immediately after the power constraint was adjusted by BM (event 7).

Observations: Event 1-6 demonstrate the efficacy and flexibility of our pre-write value guard. Two validators worked together to preemptively block write actions following policies defined by BM and the policies could be updated in real-time. Event 6-8 show the effectiveness and adaptability of our live resource monitor. It successfully aborted actions that were not blocked by the value guard but had violated the latest updated resource constraints.

D. Comparison to Existing Approaches

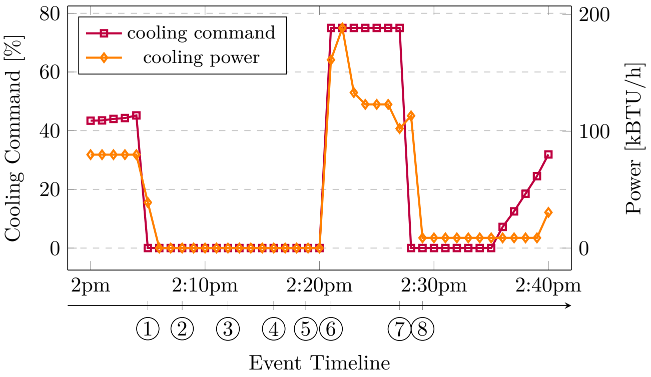


Fig. 6: AHU1 cooling conditions throughout Case Study #2. Circled numbers refer to the event number in Table II.

We here remark on limitations in existing access control and resource isolation approaches from building and computer systems domain and how aspects of our approach complement them. First, efforts to implement access control in buildings often require deep knowledge of custom ad-hoc policy languages. Nevertheless, these policies must be written on a *per-building* basis with explicit knowledge of the available resources in the building and how they relate to one another — an update on the building usually leads to an extensive rework. This micromanagement can be unrealistic in modern buildings which can contain hundreds of equipment assets and hundreds of thousands of I/O points for sensing and control. We address this issue by utilizing a structured semantic representation of the building in our proposed OS. Access control policies in PlayGround are written expressively in a standard graph query language without substantial learning costs. Moreover, these policies are generic to buildings — they can work with updated building environments or be portably reused on new buildings.

Second, most efforts do not properly address the issue of application isolation in the building domain. This is where buildings are fundamentally different from computers. Traditional OS treats shared resources as *fungible*, i.e. it does not (usually) matter which CPU cycles or which physical memory addresses an application is assigned, or when they are assigned. However, the high degree of interconnectivity between building resources means that control of one resource (e.g., a hot water coil) can influence the behavior of separate but downstream resources (e.g., the position of a valve) and the resulting effect on occupied spaces (e.g., the temperature of a room). PlayGround addresses this by allowing building managers to define arbitrary connections between resources of concern and relevant control points expressively in the format of regulating policy and resource constraints.

E. Microbenchmarks

To understand the cost of our graph-based mechanisms and the effectiveness of caching, we measured the runtime of relevant system services including capability derivation (computing capabilities of an app delegating a user), validator mapping (finding the validator queue

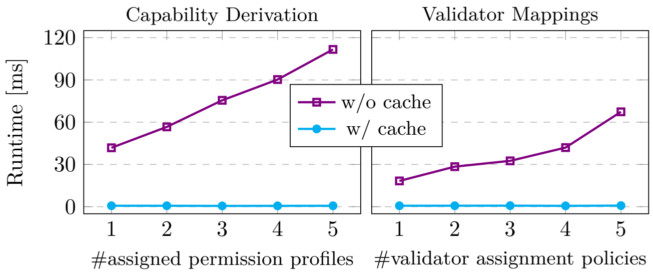


Fig. 7: Time consumption of capability derivation vs. number of permission profiles assigned to a user; and time consumption of resource validator mappings vs. number of validator assignment policies.

assigned to one resource), and resource specification retrieval (obtaining the essential metadata to read/write a resource) on the Brick graph of the same building used in case studies, with 1344 readable/writable resource points described. We also benchmark the cost of the two validators used in case study #2. The tests are executed on a machine with an AMD 3960X 24-Core CPU, RTX 3090 GPU, and 64GB memory. We measure the caching effect after cache entries have been fully pre-computed.

To benchmark capability derivation, we measure the average time consumption of the capability derivation of the same app delegating users with 1 to 5 various permission profiles assigned to them. Similarly, we obtain the average runtime of validator mapping by measuring the average cost of mapping all available resource points when there are 1 to 5 real-world validator assignment policies in the system. See Appendix E for all detailed policies (queries). We present the results with and without caching for both operations respectively in Figure 7. We can see a clear increasing pattern without caching. In contrast, caching drastically speeds up the process to <1 ms and scales well. The total memory consumption of caching both operations in full (for all the resource points under five different profiles/policies) is 4.8 MB at its peak.

We benchmark the resource specification retrieval process by measuring the average time consumption of getting the read/write specifications of a thousand randomly sampled resource points in the graph. We get an average runtime cost of 7.9 ms without caching and 0.24 ms with caching. We measure the efficiency of the range checker validator on a thousand random resources and 3 random input (writing) values and obtain an average of 9.1 ms without caching and 0.5 ms with caching. Similarly, we measure the efficiency of the power predictor validator and get an average of 1.4 ms runtime cost (with cuda). The peak memory consumption of caching these operations for all resource points is 3.3 MB. The memory consumption is reasonably low and we expect it to scale effectively with additional resources or buildings in the system.

F. Real-world Deployment

As of the time of submission, we have deployed our system on two real buildings on campus (including the one used in the case studies), beta-serving over 30 volunteer students and staff for three months consecutively. Two building applications (a power estimation app in addition to the Genie application we have mentioned above) are installed and work out of the box with no "one-off" modifications. Our resource isolation and capability mechanism successfully enables personalized zone control and building telemetry with these two (untrusted) applications while not sacrificing the safety of the building — the resource constraint set by us *e.g.,* cooling and heating power of various AHUs and maximum power consumption of smart plugs were successfully enforced. The deployment to a second building was straightforward — system policies *e.g.,* permission profiles, validator assignments, and regulating policies were directly shared and reused in these two buildings. We include all these policies we create here as template examples in the system so that future building managers with no prior knowledge of programming can get a bootstrap adopting our system. We plan to provide more ready-to-go templates for future work.

G. Limitations and Future Work

PlayGround does have limitations that we would like to address in the future. First, our system does not handle the situation where multiple users have conflicting write requests on the same resource point — we would like to explore various arbitration mechanisms to solve this problem for future work. Second, our system lacks a standardized method to enable building application compatibility checks. We plan to incorporate semantic sufficiency [31] to fill this gap. Last, we do not have a formal user study of our system with building managers though we have been working closely with them to understand the problem statements. We leave this as the subject of future work.

VI. Conclusion

We have developed PlayGround, a safe building operating system that incorporates Brick semantic representations of buildings with two proposed novel mechanisms including 1) a graph-based capability mechanism for access controls, namely permission profile; 2) a resource isolation mechanism with support of both pre-action interventions and telemetry-based live inspections. PlayGround demonstrates the feasibility of hosting untrusted, multi-tenant building applications safely while avoiding intensive maintenance and deployment costs. We accomplish this by empowering the OS services with detailed knowledge of buildings and supporting a declarative configuration setup. We envision that building managers with little or no programming background can easily adopt and maintain BOSes following our design with an emerging Brick-driven ecosystem including applications and various system policy configurations in the future.

References

[1] N. E. Klepeis, W. C. Nelson, W. R. Ott, J. P. Robinson, A. M. Tsang, P. Switzer, J. V. Behar, S. C. Hern, and W. H. Engelmann, "The national human activity pattern survey

(nhaps): a resource for assessing exposure to environmental pollutants," *Journal of Exposure Science & Environmental Epidemiology*, vol. 11, no. 3, pp. 231–252, 2001.

[2] U. Energy Information Administration (EIA), "Global energy consumption driven by more electricity in residential, commercial buildings — eia.gov." https://www.eia.gov/todayinenergy/detail.php?id=41753, 2023.

[3] S. Dawson-Haggerty, A. Krioukov, J. Taneja, S. Karandikar, G. Fierro, N. Kitaev, and D. Culler, "{BOSS}: Building Operating System Services," pp. 443–457, 2013.

[4] T. Weng, A. Nwokafor, and Y. Agarwal, "BuildingDepot 2.0: An Integrated Management System for Building Analysis and Control," in *Proceedings of the 5th ACM Workshop on Embedded Systems For Energy-Efficient Buildings*, (Roma Italy), pp. 1–8, ACM, Nov. 2013.

[5] M. P. Andersen, J. Kolb, K. Chen, G. Fierro, D. E. Culler, and R. Katz, "Democratizing Authority in the Built Environment," *ACM Transactions on Sensor Networks*, vol. 14, pp. 1–26, Dec. 2018.

[6] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl, "An Operating System for the Home," pp. 337–352, 2012.

[7] G. Fierro, M. Pritoni, M. Abdelbaky, D. Lengyel, J. Leyden, A. Prakash, P. Gupta, P. Raftery, T. Peffer, G. Thomson, and D. E. Culler, "Mortar: An open testbed for portable building analytics," *ACM Trans. Sen. Netw.*, vol. 16, dec 2019.

[8] F. He, Y. Deng, Y. Xu, C. Xu, D. Hong, and D. Wang, "Energon: A data acquisition system for portable building analytics," in *Proceedings of the Twelfth ACM International Conference on Future Energy Systems*, e-Energy '21, (New York, NY, USA), p. 15–26, Association for Computing Machinery, 2021.

[9] A. Krioukov, G. Fierro, N. Kitaev, and D. Culler, "Building application stack (bas)," in *Proceedings of the Fourth ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*, BuildSys '12, (New York, NY, USA), p. 72–79, Association for Computing Machinery, 2012.

[10] X. Fu, J. Koh, F. Fraternali, D. Hong, and R. Gupta, "Zonal air handling in commercial buildings," in *Proceedings of the 7th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, BuildSys '20, (New York, NY, USA), p. 302–303, Association for Computing Machinery, 2020.

[11] Y. Agarwal, B. Balaji, S. Dutta, R. K. Gupta, and T. Weng, "Duty-cycling buildings aggressively: The next frontier in HVAC control," in *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks*, pp. 246–257, Apr. 2011.

[12] Y. Agarwal, B. Balaji, R. Gupta, J. Lyles, M. Wei, and T. Weng, "Occupancy-driven energy management for smart building automation," in *Proceedings of the 2nd ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Building - BuildSys '10*, (Zurich, Switzerland), p. 1, ACM Press, 2010.

[13] B. Balaji, J. Koh, N. Weibel, and Y. Agarwal, "Genie: a longitudinal study comparing physical and software thermostats in office buildings," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '16, (New York, NY, USA), pp. 1200–1211, Association for Computing Machinery, Sept. 2016.

[14] B. Balaji, A. Bhattacharya, G. Fierro, J. Gao, J. Gluck, D. Hong, A. Johansen, J. Koh, J. Ploennigs, Y. Agarwal, M. Bergés, D. Culler, R. K. Gupta, M. B. Kjærgaard, M. Srivastava, and K. Whitehouse, "Brick : Metadata schema for portable smart building applications," *Applied Energy*, vol. 226, pp. 1273–1292, Sept. 2018.

[15] "Project haystack," 2022. https://project-haystack.org/.

[16] ASHRAE, "ASHRAE's BACnet Committee, Project Haystack and Brick Schema Collaborating to Provide Unified Data Semantic Modeling Solution." http://tinyurl.com/7c9yxbn3, 2018.

[17] K. Hammar, E. O. Wallin, P. Karlberg, and D. Hälleberg, "The realestatecore ontology," in *The Semantic Web – ISWC 2019: 18th International Semantic Web Conference, Auckland, New Zealand, October 26–30, 2019, Proceedings, Part II*, (Berlin, Heidelberg), p. 130–145, Springer-Verlag, 2019.

[18] K. Berkoben, C. E. Kaed, and T. Sodorff, "A digital buildings ontology for google's real estate," in *International Workshop on the Semantic Web*, 2020.

[19] J. Koh, D. Hong, S. Nagare, S. Boovaraghavan, Y. Agarwal, and R. Gupta, "Who can Access What, and When?: Understanding Minimal Access Requirements of Building Applications," in *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, (New York NY USA), pp. 121–124, ACM, Nov. 2019.

[20] M. P. Andersen, S. Kumar, M. AbdelBaky, G. Fierro, J. Kolb, H.-S. Kim, D. E. Culler, and R. A. Popa, "{WAVE}: A Decentralized Authorization Framework with Transitive Delegation," pp. 1375–1392, 2019.

[21] P. Arjunan, M. Saha, H. Choi, M. Gulati, A. Singh, P. Singh, and M. B. Srivastava, "SensorAct: A Decentralized and Scriptable Middleware for Smart Energy Buildings," in *2015 IEEE 12th Intl Conf on Ubiquitous Intelligence and Computing and 2015 IEEE 12th Intl Conf on Autonomic and Trusted Computing and 2015 IEEE 15th Intl Conf on Scalable Computing and Communications and Its Associated Workshops (UIC-ATC-ScalCom)*, (Beijing), pp. 11–19, IEEE, Aug. 2015.

[22] S.-H. Leitner and W. Mahnke, "Opc ua–service-oriented architecture for industrial applications," *Softwaretechnik-Trends Band 26, Heft 4*, 2006.

[23] M. Pritoni, D. Paine, G. Fierro, C. Mosiman, M. Poplawski, A. Saha, J. Bender, and J. Granderson, "Metadata schemas and ontologies for building energy applications: A critical review and use case analysis," *Energies*, vol. 14, no. 7, 2021.

[24] A. A. Bhattacharya, D. Hong, D. Culler, J. Ortiz, K. Whitehouse, and E. Wu, "Automated metadata construction to support portable building applications," in *Proceedings of the 2nd ACM International Conference on Embedded Systems for Energy-Efficient Built Environments*, BuildSys '15, (New York, NY, USA), p. 3–12, Association for Computing Machinery, 2015.

[25] H. Bergmann, C. Mosiman, A. Saha, S. Haile, W. Livingood, S. Bushby, G. Fierro, J. Bender, M. Poplawski, J. Granderson, and M. Pritoni, "Semantic interoperability to enable smart, grid-interactive efficient buildings," 12 2020.

[26] K. Janowicz, M. H. Rasmussen, M. Lefrançois, G. F. Schneider, and P. Pauwels, "Bot: The building topology ontology of the w3c linked building data group," *Semant. Web*, vol. 12, p. 143–161, jan 2021.

[27] J. Koh, D. Hong, R. Gupta, K. Whitehouse, H. Wang, and Y. Agarwal, "Plaster: An integration, benchmark, and development framework for metadata normalization methods," in *Proceedings of the 5th Conference on Systems for Built Environments*, BuildSys '18, (New York, NY, USA), p. 1–10, Association for Computing Machinery, 2018.

[28] G. Fierro, A. K. Prakash, C. Mosiman, M. Pritoni, P. Raftery, M. Wetter, and D. E. Culler, "Shepherding metadata through the building lifecycle," in *Proceedings of the 7th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, BuildSys '20, (New York, NY, USA), p. 70–79, Association for Computing Machinery, 2020.

[29] H. Lange, A. Johansen, and M. B. Kjærgaard, "Evaluation of the opportunities and limitations of using ifc models as source of building metadata," in *Proceedings of the 5th Conference on Systems for Built Environments*, BuildSys '18, (New York, NY, USA), p. 21–24, Association for Computing Machinery, 2018.

[30] F. He and D. Wang, "Cloze: A building metadata model generation system based on information extraction," in *Proceedings of the 9th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, BuildSys '22, (New York, NY, USA), p. 109–118, Association for Computing Machinery, 2022.

[31] G. Fierro, A. Saha, T. Shapinsky, M. Steen, and H. Eslinger, "Application-driven creation of building metadata models with semantic sufficiency," in *Proceedings of the 9th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, BuildSys '22, (New York, NY, USA), p. 228–237, Association for Computing Machinery, 2022.

```
ex:plug1 a brick:Active_Power_Sensor;
   brick:hasUnit unit:MilliW ;
   ref:hasExternalReference [
      a ref:BACnetReference ;
      bacnet:object-identifier "181160_analogInput_3" ;
      bacnet:object-name "AI3 BERT Power Measurement" ;
      bacnet:objectOf ex:plug1_BACnetDevice ; ] ;
   ref:hasTimeseriesReference [
      a ref:TimeseriesReference ;
      ref:hasTimeseriesId
      ↪  "0000000-0000-000-0000-000000000000" ;
      ref:storedAt "postgres://1.2.3.4:5432/db1" ; ] .
```

Fig. 8: An example Brick resource representation with external references.

Appendix

A. Implementation of Resource Proxy

We store the necessary specification of a resource point to support read/write operations as external reference objects in Brick. Resource Proxy retrieves this information from the graph given the Brick reference and finds the corresponding interface to carry out the request. We present an example in Figure 8. Provided with the Brick reference `ex:plug1`, Resource Proxy first runs a SPARQL query to find the type of the external reference, which in this case is `ref:BACnetReference`. Resource Proxy then invokes the driver interface for this resource type with the payload plus other remaining specifications in the external reference (object identifier and object upstream in this example). When a read request is accompanied by a time range in its payload, it indicates a time-series read request. In this case, Resource Proxy will look for a `ref:TimeseriesReference` object in particular and invoke the time-series driver. Any failure encountered in the above process will abort this incoming read/write request.

Various driver interfaces for distinct types of resources all follow the same signature by inheriting a provided abstract base class (ABC) — take as input the resource endpoint plus the payload and perform operations accordingly. For instance, when there's a True relinquish flag in the payload of a write request, it should release or reset the control over the resource (if possible). To extend support to new types of resources, we only need to import new driver interfaces inheriting this ABC to our kernel program.

B. Implementation of App Steward

All app containers reside in a separate docker network, where we enforce strict network restrictions as a whitelist-based "firewall". By default, the only destination on the whitelist that an app container can communicate with is the core container hosting the API interface — the external endpoints listed in the app manifest are added to the whitelist respective to the app. App Steward dynamically maintains these whitelists by manipulating **`iptables`** on this docker network interface. App Steward also interacts with the underlying docker daemon to manage each app container instance.

1) Registration: When the registration of an application is approved, App Steward stores the provided app manifest and permission profile (find more details on this in §C), saves the frontend static files, and builds a docker image for the backend (we require the existence of a valid dockerfile in the source code).

2) Execution: When a user asks to instantiate an application, App Steward starts a docker container with the built image of the application and loads the frontend static files. It also passes a `JWT` token signed and generated for this app instance to the container as an environment variable so that the app instance can use it with the APIs. Meanwhile, App Steward stores the essential metadata associated with this app instance such as container id, the initiating user, and the argument for the permission profile provided by the user. Applications can call an API to get a list of resources from which a user can select, for instance, to specify the room or equipment. This follows the same rule as the argument that they can provide for the app's permission profile.

3) Termination: When an application instance needs to exit or be gracefully terminated, App Steward always tries to stop the container with the associative container id gracefully with a SIGTERM signal. Applications are expected to react to SIGTERM promptly and perform necessary clean-ups and relinquish action. If the container fails to exit within a 30 s timeout or it is forcefully terminated, a force stop (SIGKILL) will be enforced. A stopped docker container is typically not removed for faster start-up in the future.

C. Implementation of Permission Manager

The core of Permission Manager is the permission profiles. We store the SPARQL queries of each permission profile in f-string format — when invoked, the associated arguments will be filled in. Permission Manager maintains the user permission profile assignment table as well. The capability derivation procedure described in §III-D in practice would be 1) Permission Manager first looks up the user permission profile assignment table and computes the capability of the user given the stored SPARQL queries and associated arguments; 2) then it uses the arguments for the permission profile of the app saved by App Steward per instantiation to compute the raw app capability; 3) it derives the final capability with the delegation scheme specified in the app manifest.

D. Resource Regulator

1) Resource Constraint Storage: Resource constraint storage provides an interface to enquire about whether a particular resource value exceeds the constraint or not, preventing other kernel components from raw queries to the resource constraint table

a) Updated `Grad_student`

```
Def Grad_Student(room: brick:Room)
---------
READ+WRITE
SELECT ?p WHERE {
?e brick:hasLocation {room} .
?p brick:isPointOf ?e .
?p a ?o .
FILTER (?o IN
   (brick:TemperatureSetpoint,
   brick:On_Off_Command))
}
READ
SELECT ?p WHERE {
?e brick:hasLocation {room} .
?e brick:hasPoint ?p .
?p a ?o .
FILTER (?o IN
   (brick:TemperatureSensor,
   brick:OccupancySensor))
}
```

b) Permission profile for lab managers

```
Def Lab_Manager(room: brick:Room)
---------
READ+WRITE
SELECT ?p WHERE {
?e brick:hasLocation/brick:feeds*
↪  {room} .
?e a brick:Equipment .
?p brick:isPointOf ?e . }
READ
NULL
```

c) Permission profile plug masters

```
Def Plug_Master(plug:
↪  brick:Smart_Plug)
---------
READ+WRITE
SELECT ?p WHERE {
?p brick:isPointOf {plug} .}
READ
NULL
```

Fig. 9: Sample permission profiles we used in case study #1 and the microbenchmark for permission derivation.

a) “First Floor” Policy

```
SELECT DISTINCT ?p WHERE {
   ?loc brick:isPartOf
   ↪  ex:First_Floor .
   ?equip brick:feeds ?loc .
   ?p brick:isPointOf ?equip .
}
```

b) “AH-4” Policy

```
SELECT DISTINCT ?p WHERE {
   {?p a brick:Point .
   ?p brick:isPointOf ?equip .
   ex:AH-4 brick:feeds ?equip .}
   UNION
   {?p a brick:Point .
   ?p brick:isPointOf ex:AH-4 .}
}
```

c) “VAV-2” Policy

```
SELECT DISTINCT ?p WHERE {
   ?p a brick:Point .
   ?p brick:isPointOf ex:VAV-2 .
}
```

d) “Third Floor” Policy

```
SELECT DISTINCT ?p WHERE {
   ?loc brick:isPartOf
   ↪  ex:Third_Floor .
   ?equip brick:feeds* ?loc .
   ?p brick:isPointOf ?equip .
}
```

e) “Default” Policy

```
SELECT DISTINCT ?equip WHERE {
   ?equip a brick:Point .
}
```

Fig. 10: Sample validator assignment policies we used in the microbenchmark for validator mappings.

2) Pre-actuation Value Validators: Validators, similar to driver interfaces in Resource Proxy, inherit an ABC to make sure that they follow the same abstraction as described in §III-E. We can add new validators to the system by simply importing more child classes. In practice, example validators can be prediction functions on energy consumption or some basic range checkers on the write values.

3) Validator Assignments: Recall that building managers may use SPARQL queries to assign validators to a set of building resources. Unlike permission profiles, the lookup process to find the validators assigned to a resource here is reversed — we need to find which set of resources (or SPARQL-defined policy) includes the resource under questioning and obtain the validator queue accordingly. However, query-defined sets are not necessarily mutually exclusive. Therefore we allow building managers to optionally define the priority of each policy. Policies of higher priority are looked up first; the order of policies of the same priority being looked up is random. Once a policy is found to contain the target resource, the lookup process is finished.

4) Regulating Policies: Again, regulating policies inherit an ABC and follow the abstraction that takes as input a pair of a resource and the current value of this resource (exceeding the constraint), and decide follow-up actions accordingly. There's no return value. The exact behavior of a regulating policy is highly customizable. With the input provided, an example regulating policy may first query the Brick Oracle to figure out the attributing resources and the transaction history within Resource Proxy to obtain a history of which application on behalf of which user made what actuation at when to help identify the guilty applications. Then it may decide to terminate these applications and relinquish these control points by interacting with the App Steward and Resource Proxy.

E. Sample Policies Used

The updated permission profile for `Grad_student` used in case study #1 is shown in Figure 9.a. The other permission profiles we used to be randomly assigned to users in the microbenchmark for permission derivation are listed in the rest of Figure 9. The `Lab_Manager` permission profile is defined for lab managers who should have write access to all equipment points in the assigned room/lab. The `Plug_Master` permission profile will grant one user write access to all points of the assigned smart plug (`brick:Smart_Plug` is a customized extension to Brick not included in the official Brick ontology).

The five validator assignment policies we used in the microbenchmark for validator mappings are listed in Figure 10. “First Floor” defines the set composed of all points of equipment that feed sub-locations on the first floor of the building. “AH-4” defines the set of resources consisting of all points of AH-4 and its sub-components. “VAV-2” defines the set of all points VAV-2. “Third Floor” is similar to “First Floor” but applies to the third floor. “Default” is the fallback policy which simply contains all points in this building. In the microbenchmark, we rank their priority in the following order: “Third Floor”, “VAV-2”, “AH-4”, “First Floor”, and “Default”. When there are fewer than five policies in the system in the experiment, we keep the less prioritized ones. For instance, when the number of assignment policies in the system is 3 (*i.e.,* the third datapoint for validator mappings in Figure 7), we have them ordered as “AH-4”, “First Floor”, and “Default”.