# Aerogel: Lightweight Access Control Framework for WebAssembly-Based Bare-Metal IoT Devices

Renju Liu
University of California, Los Angeles
Los Angeles, CA, USA

Luis Garcia
University of Southern California
Information Sciences Institute
Marina Del Rey, CA, USA

Mani Srivastava
University of California, Los Angeles
Los Angeles, CA, USA

## ABSTRACT

Application latency requirements, privacy, and security concerns have naturally pushed computing onto smartphone and IoT devices in a decentralized manner. In response to these demands, researchers have developed micro-runtimes for WebAssembly (Wasm) on IoT devices to enable streaming applications to a runtime that can run the target binaries that are independent of the device. However, the migration of Wasm and the associated security research has neglected the urgent needs of access control on *bare-metal*, memory management unit (MMU)-less IoT devices that are sensing and actuating upon the physical environment. This paper presents Aerogel, an access control framework that addresses security gaps between the bare-metal IoT devices and the Wasm execution environment concerning access control for sensors, actuators, processor energy usage, and memory usage. In particular, we treat the runtime as a multi-tenant environment, where each Wasm-based application is a *tenant*. We leverage the inherent sandboxing mechanisms of Wasm to enforce the access control policies to sensors and actuators without trusting the bare-metal operating system. We evaluate our approach on a representative IoT development board: a cortex-M4 based development board (nRF52840). Our results show that Aerogel can effectively enforce compute resource and peripheral access control policies while introducing as little as 0.19% to 1.04% runtime overhead and consuming only 18.8% to 45.9% extra energy.

## CCS CONCEPTS

• **Security and privacy → Software security engineering**; **Mobile platform security**.

**ACM Reference Format:**
Renju Liu, Luis Garcia, and Mani Srivastava. 2021. Aerogel: Lightweight Access Control Framework for WebAssembly-Based Bare-Metal IoT Devices. In *The Sixth ACM/IEEE Symposium on Edge Computing (SEC '21), December 14–17, 2021, San Jose, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3453142.3491282

## 1 INTRODUCTION

The scope of leveraging mobile and internet-of-things (IoT) devices for sensing physical spaces has generalized beyond human activity recognition. Distributed and deployed IoT systems leverage the ubiquitous sensors for a myriad of applications such as smart healthcare, smart city lighting, and transportation [25]. In the 5G context, on-device edge computation enables nascent market applications such as augmented reality, mass IoT, and drone services [53]. Consequently, the latency requirements, privacy, and security concerns for these safety-critical applications have naturally shifted computation from centralized cloud resources to decentralized edge IoT and mobile devices [50].

The heterogeneity of the underlying device hardware and software ecosystems poses complex challenges for application developers. The dynamicity and heterogeneity of these devices necessitate the support for dynamically instantiated, portable workloads stemming from more than one source while maintaining security and performance for applications. More critically, the isolation mechanisms to secure these platforms assume some form of memory management unit (MMU) [7]. Several resource- and energy-constrained IoT hardware platforms do not support MMUs [5, 6]. For instance, in 2016, experts estimated ARM to have shipped 22 billion units of the MMU-less Cortex-M based devices [19]. Although Cortex-M processors are enabled with *Memory Protection Units (MPU)* that can provide memory isolation, MPUs can only support a finite number of memory regions. Moreover, applications need to be rewritten under different bare-metal OSes that use MPUs because they require different OS abstractions [10].

**Latest advances.** Recent works have only partially addressed the requirements of security, performance, portability, and dynamic instantiation for heterogeneous, resource-constrained computation. First, recent solutions that focus on memory protection for bare-metal devices [3, 17, 37, 46, 52] almost always rely on MPU–assuming that applications will be re-written for different OSes. Similarly, secure formal verification-based microkernels such as SeL4 [27] target more powerful processors such as the ARM A-series platforms due to the performance and design restrictions. Moreover, these solutions typically ignore IoT devices' cyber-physical nature and do not provide sufficient access control to peripherals, i.e., sensors and actuators. The shortfall of these solutions resides in the fact that they were not designed for multi-tenant and heterogeneous applications within distributed environments. Further, static security or ad-hoc policy implementations [13, 14, 35, 49] fall short since the application does not need to follow the policy at runtime if the policy enforcement mechanism does not exist–which is where Aerogel is expected to fill that gap.

**Challenges.** We summarize the three interdependent research challenges as follows. First, how do we provide fine-grained memory protection for *multi-tenant* IoT devices? Second, how can we ensure


*SEC '21, December 14–17, 2021, San Jose, CA, USA*

ACM ISBN 978-1-4503-8390-5/21/12…$15.00
https://doi.org/10.1145/3453142.3491282

access to sensors and actuators are protected within multi-tenant applications? Third, given time- and resource-constrained IoT devices, how do we provide such security mechanisms while maintaining a minimal overhead and memory footprint in real-time?

**Wasm for IoT.** The requirements of security, performance, portability, and dynamic instantiation for heterogeneous computation platforms are not unique to the 5G IoT edge. In response to the increasing demands of performance and security for web application deployment in fragmented and heterogeneous environments, engineers from the top four major browser vendors collaboratively developed WebAssembly (Wasm) [16]–a portable low-level bytecode that is platform-independent. Subsequently, industry giants such as Intel and Redhat formed an alliance, known as the Bytecode Alliance [12], to develop a micro-runtime for Wasm that is supported by *bare-metal* IoT devices, i.e., resource-constrained, MMU-less devices where all software models share the same memory space. The Wasm Micro-runtime (WAMR) [11] enables applications that can run the target binary at native speeds independent of the device and can achieve sandboxing without a memory-management unit.

Although WAMR is a strong candidate to support secure, performant, and *multi-tenant* computation on edge, the scope of IoT applications is not limited to computation services. The computation abstractions will run alongside sensing and actuation services provided by the IoT device that interacts with the physical world. WAMR currently lacks the abstractions necessary to provide access control to sensors and actuators for IoT devices while maintaining performance and security. Steps have been made towards providing limited access control (e.g., only for certain memory regions or pieces of sensitive information) for multi-tenant IoT devices using hypervisors [9, 23, 34], using the compiler at the compilation time [13, 14, 35], or using secure runtime memory views based on offline static analysis [55]. However, the proposed architectures are device-dependent, requiring the recoding and recompilation of the software stack for different IoT device architectures. The shifting workloads of the dynamic and heterogeneous IoT edge will require over-the-air (OTA) updates at runtime while supporting other *tenants*. Thus, in this paper, we aim to tackle the following challenge: how can we extend the security capabilities of Wasm on IoT to include access control for multi-tenant IoT device peripherals while maintaining performance and low resource overhead?

**Approach.** In this paper, we design Aerogel, a runtime framework that utilizes the protection mechanisms of Wasm bytecode sandboxing to provide access protection for IoT device peripherals – even when the applications and the OS are sharing the same address space. Aerogel builds upon the Wasm runtime to provide micro-management for each tenant (application). Tenant applications are compiled into Wasm bytecode such that applications can be platform-independent. The Wasm runtime isolates application bytecode from any platform-dependent native code that needs to interact with the application. Aerogel instruments Wasm runtime to provide a fine-grained access control mechanism such that users can easily define the processor energy consumption, memory usage, as well as access to sensor and actuator peripherals for each application. Moreover, our approach ensures that the applications can be regulated based on the user's security context while they run on the same address space as the OS.

We evaluate Aerogel on a low-power, resource-constrained MCU dev board (nRF52840) and benchmark a representative set of safety-critical IoT applications. Aerogel's runtime overhead ranges from 0.19% to 1.04% extra execution time and from 18.8% to 45.9% extra energy on our proposed benchmarks. Our results show that the fine-grained access control mechanism provides minimal overhead for MCU energy, and peripheral access energy while having a minimal overhead on application execution relative to related works.

**Contributions.** We summarize our contributions as follows.

- We propose Aerogel, a Wasm-based access control mechanism for bare-metal IoT devices. Wasm enables platform-independent application execution necessary for heterogeneous IoT networks.
- Aerogel leverages the sandboxing capabilities of Wasm to isolate tenant applications from each other as well as from platform-dependent native code. Aerogel enables secure sandboxing for multi-tenant applications for resource-constrained (less than 1 MB of memory), low-power devices.
- We evaluate Aerogel on a real low-power, resource-constrained MCU and show results of minimal 0.39$\mu$Ah extra energy and minimal overhead 2.1ms.

The rest of the paper is organized as follows. Section 2 briefly discusses the background information of Wasm and bare-metal IoT devices. We then overview Aerogel in section 3 and explain the details of the design in section 4. We talk about the implementation in section 5 and evaluate our work in section 6. We next analyze the security issues and limitations and discuss the future work in section 7. We compare Aerogel with the related work in section 8. Lastly, we conclude this paper in section 9.

Our source code is open-source and available online[1][2].

## 2 BACKGROUND

We first discuss the emerging field of multi-tenancy on bare-metal IoT devices. We then describe the security guarantees provided by Wasm and give a brief overview of the Wasm runtime for IoT devices.

### 2.1 Multi-tenant Bare-metal IoT Devices

*Bare-metal* IoT devices have shifted away from single-purpose applications as equipped sensors and actuators enable them to perform multiple tasks. For example, the battery-powered smart camera Blink XT2 [51] can capture images and perform on-device object detection. Further, the development ecosystem of IoT devices has enabled APIs for developers to implement applications that leverage the sensor and actuator abstractions, e.g., the Skills API for Amazon Alexa [2]. Hence, we model the complex and fragmented software and hardware IoT ecosystems as *multi-tenant* application environments. However, supporting multi-tenancy confounds the challenges of performance, sustainability, and security on resource-constrained devices.

[1] https://github.com/nesl/wasm-trustzone.git
[2] https://github.com/nesl/zephyr-wasm.git

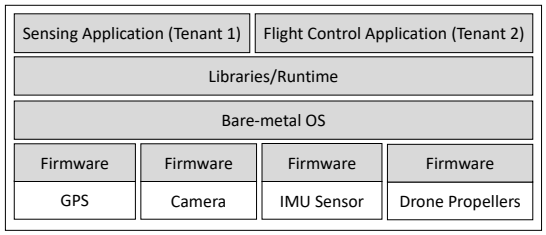


**Figure 1: The UAV example of multi-tenant bare-metal software stacks Aerogel targets on. All the code in the shaded areas runs on the same address space.**

**Bare-metal characterization.** We characterize *bare-metal* IoT devices with limited resources, such as small battery capacities containing few thousand mAh energy, low-end microprocessors (MCUs) with only a few hundred MHz frequencies, or small memory size with few hundred Kilo-Byte(KB) memories. These devices typically do not have complicated memory protection mechanisms such as user space and kernel space address separation through the Memory Management Unit (MMU). Moreover, those devices are designed for heterogeneous sensing and actuation workloads such as Umanned Aerial Vehicles(UAVs) and smart home sensors. For example, Pixhawk 4 [18] flight control device is equipped with two ARM-M processors that have 216 MHz for flight control and 24 MHz MCUs for I/O operations, and each processor has 512KB and 8KB RAM respectively. Other popular bare-metal IoT device examples include the Nest Protect [22] (MCU = 100 MHz [5] and RAM = 512 KB) and the EdgeReady Voice Control platform [44] (MCU = 600 MHz [6] and RAM = 1024 KB).

**Lack of multi-tenant isolation.** MMU-less, bare-metal IoT devices cannot provide memory isolation among different applications. Figure 1 shows an example of a bare-metal UAV system supporting two different applications, i.e., tenants, that perform sensing tasks to control flight dynamics. If various entities develop the applications, the bare-metal devices would not be able to protect one safety-critical application from another application's bugs or vulnerabilities. Although researchers [17, 37, 46] have proposed to leverage Memory Protection Units (MPUs) on ARM Cortex-M based IoT devices to provide memory isolation [13], MPUs can only support a finite number of memory regions. Moreover, the associated applications would not be portable as they need to be rewritten under different bare-metal OSes. Thus, we require a lightweight, portable, and software-based memory isolation for multi-tenant applications.

## 2.2 WebAssembly for Non-web Embeddings

Researchers have adopted WebAssembly [16] (Wasm) to account for the bottlenecks of security, portability, and dynamic instantiation. Wasm was initially designed for web browser JavaScript applications on heterogeneous client devices to enhance the security of script isolation, improve web application execution speed, and erase the execution environment's heterogeneity requirement. Wasm has since generalized beyond web embeddings to bare-metal IoT devices due to the original problems' generality.

*2.2.1 Tenets of Wasm.* We first motivate the major advantages of Wasm in the context of bare-metal IoT devices.

**Performance.** Wasm is designed as a statically-typed programming language such that the variable type is determined at compilation time. Additionally, Wasm utilizes a linear memory structure that is loaded as a compact binary format. Hence, Wasm is able to achieve near-native speed performance [1, 24].

**Security.** Wasm can provide runtime code isolation for applications under any known memory-layout architecture (e.g., Arm M-series boards) by running them in sandboxed execution environments. Inside each sandbox, the Wasm application has full access to its memory. However, any access to the memory outside of the sandbox throws an exception. Moreover, each Wasm application has its own sandbox, and its sandbox cannot be arbitrarily accessed by different applications. By achieving this, Wasm runtime linearly allocates memory regions and ensures that the entry and the exit point of a function do not go beyond the sandbox bounds. If a Wasm bytecode instruction reads from or writes to a specific memory address, Wasm runtime will check whether the memory address is within the application's sandboxed memory regions.

**Portability and Dynamic Instantiation.** Wasm is a platform-independent binary format whose execution resides on top of its runtime. Hence, Wasm applications are portable on any device that has a Wasm runtime and can initiate the execution environment without recompiling the software stack.

*2.2.2 Wasm Micro-runtime.* Wasm bytecode is executed on a Wasm runtime. From the runtime's perspective, Wasm bytecode is a group of Wasm bytecode instructions, where each instruction is encoded with one OPCODE followed by one or more arguments. For example, `i32.add(i32.const 3)(i32.const 2)` computes the addition of 3 and 2. Moreover, Wasm byte code applications have a special instruction that can execute native functions inside Wasm runtime exported through pre-registered function tables.

Wasm micro-runtime [11] (WAMR) is one of the most popular lightweight runtimes for Wasm bytecode on bare-metal devices. WAMR has only a few hundred kilobytes of memory footprint. WAMR manages all the execution of Wasm bytecode. At the beginning, it allocates a contiguous memory region for the Wasm application that can only grow contiguously starting from the end address. Next, each Wasm instruction is translated into machine code by WAMR. To reduce the execution overhead, WAMR also allows a mixture of Wasm bytecode execution and platform-dependent execution. A mixture may be allowed for platform-dependent functionality optimization. To maintain security enforcement, the associated native code is provided by WAMR and not the application developer. Although WAMR provides the initial framework for Wasm on IoT devices, Aerogel will aim to provide a peripheral access control framework for bare-metal IoT applications. Next, we overview the design of Aerogel.

# 3 OVERVIEW

We first describe the threat model and goals of Aerogel followed by the design workflow.

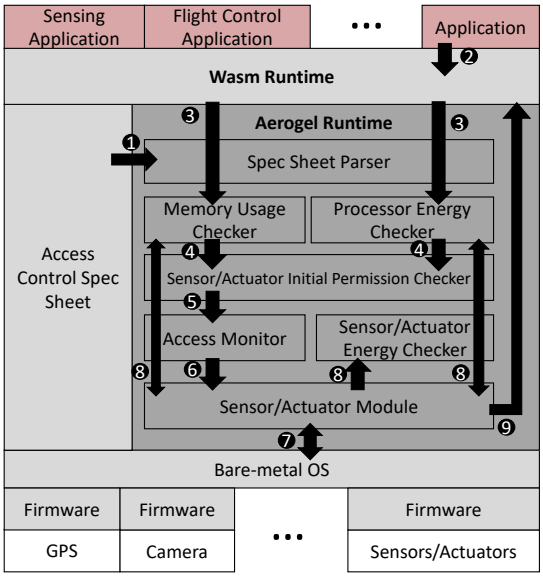


**Figure 2: An overview and the workflow of Aerogel. The darker grey area is the component of Aerogel.**

## 3.1 Threat Model and Assumptions

Aerogel trusts the software stack below the applications running on bare-metal IoT devices. More specifically speaking, Aerogel trusts the firmware, the bare-metal OS, and the Wasm runtime. Aerogel does not trust any application. We assume the entire software stack code–including the application, the bare-metal OS, the Wasm runtime, and the firmware of the hardware–is running on the same address space as there is no MMU for the memory address space separation. Any memory attacks such as buffer-overflow performed by a malicious application to obtain the unauthorized access to sensors or actuators can be prevented. Moreover, Aerogel also protects against attacks of improper energy usage after the application is authorized to access a sensor or an actuator. Side-channel attacks, including cyber-physical attacks, towards the sensors or actuators such as GPS spoofing are out of the scope of this paper. Moreover, we assume sensor and actuator abstractions are provided to allow multiple applications to use the sensors or actuators simultaneously, e.g., as was done in prior works [38, 39].

## 3.2 Goals

We enumerate the design goals for Aerogel as follows:

- **Sensor and actuator access protection**. Each application is expected to be isolated from each other under a robust sandboxed execution environment. Aerogel needs to make sure its execution environment does not allow arbitrary access to the peripherals such as sensors or actuators.
- **Fine-grained access control**. Aerogel aims to provide a access control mechanism such that the users are able to define the the processor energy consumption, memory usage, as well as the energy consumption per sensor and actuator for each application.
- **Minimal overhead and memory footprint**. Aerogel aims to provide low overhead and low memory footprint runtime support such that it has minimal execution impact for all applications on the resource-constrained bare-metal devices.

## 3.3 Workflow

Figure 2 provides an overview of Aerogel's design workflow. Prior to execution, all application code needs to be compiled to Wasm bytecode. At runtime, Aerogel parses the access control specification sheet (❶), which is the user-defined access rules whose details are discussed in section 4.2. After the specification sheet is processed, the Wasm runtime loads the Wasm bytecode of the applications as Wasm instructions and initiates their runtime environment(❷). Wasm runtime then attempts to execute the loaded Wasm instructions. Upon each instruction execution, the Wasm runtime makes a request to the Aerogel runtime checker to determine if the current application has exceeded the maximum allowable processor energy and memory usage(❸). If an application's processor usage has exceed its allotment (assuming the allotment exists), Aerogel will request the bare-metal OS schedule the current application to sleep for a user-defined period of time. The total processor energy consumption for the particular application is reset to zero by the processor energy checker after the user-defined reset time in the access control specification sheet has elapsed. Resetting the application's energy consumption prevents the application from sleeping indefinitely. If the application's memory usage has reached its allotment, the memory can no longer be increased.

If the Wasm instruction requires reading sensor data or writing data to actuator peripherals, a request is sent to Aerogel's initial sensor and actuator permission checker to check whether such an access is allowed in the user-defined sensor and actuator allowlist (❹). Once the initial access has been cleared, Aerogel's runtime checks whether the maximum total number of accesses has been achieved for the requested sensors or actuators by the access monitor(❺). If either the initial permission checking or the maximum number of access checking fails, the access is denied. Aerogel's runtime throws an exception that is handled by Wasm runtime. Otherwise, the request is passed to be registered by the sensor and actuator module(❻) that directly interfaces with the sensors and actuators. When new sensor data for or a new actuation command from a particular application needs to be handled by the sensor and actuator module(❼), the module sends requests to the energy usage and the memory usage checkers to verify whether the memory usage or the energy usage has exceeded the maximum allotment for the associated sensor or actuator (❽). If not, the sensor and actuator module executes the actuation command or sends back the new sensor data to the application (❾). Otherwise, the corresponding command or data is discarded.

We next explain the details of Aerogel's design.

# 4 AEROGEL RUNTIME

We first describe how Aerogel provides memory protection for sensors and actuators. We then explain how access control policies are defined and enforced.

## 4.1 Wasm-based Peripheral Memory Isolation

Aerogel's bare-metal peripheral access control hinges on isolating the peripheral memory locations from the application memory that resides on the same contiguous linear memory space. We describe how Aerogel isolates the peripherals from application memory in two stages: application instantiation and application runtime.

*4.1.1 Isolation at Application Instantiation.* When the Wasm runtime instantiates the runtime environment for an application, it needs to allocate the associated memory heap. The runtime searches for the first available memory region from the beginning of the linear physical memory. Aerogel's runtime checks whether such allocation has overlapped with sensor or actuators addresses, i.e., by checking whether the linear regions include sensor or actuator addresses. If an overlap is detected, Aerogel's runtime returns the first possible available memory regions that do not overlap with the I/O address of the sensors and actuators with the required size of the memory and returns the start address to Wasm runtime.

**Memory collision resolution.** When Wasm runtime's requested memory region overlaps with a sensor's or actuator's I/O address that instantiates the abstractions of the corresponding I/O devices, Aerogel's runtime starts looking for the first possible memory addresses that could fulfill with the request. Aerogel's runtime first searches from the low address to the high address of all sensors and actuators without considering other conditions such as whether the memory is used by other applications, further checked by the Wasm runtime. For instance, a sensor and an actuator may have addresses `0x8FFFFFE0` and `0x8FFFFFF0` respectively, and the Wasm runtime needs to allocate `0x100` bytes of memory. Assuming the memory is aligned in 4-byte settings, Aerogel's runtime first checks whether `0x8FFFFFD4` can fulfill with the request of `0x100` memory size. In this instance, the allocation overlaps with the actuator's address (`0x8FFFFFE0`). Aerogel's runtime then checks whether `0x8FFFFFE4` can be a potential candidate and ensures there are no other actuators or sensors between `0x8FFFFFE4` and `0x900000D0` (`0x8FFFFFE4` + `0x100`), hence `0x8FFFFFE4` will be returned to Wasm runtime that will do further checks of whether the memory regions will be eligible.

*4.1.2 Isolation at Application Runtime.* When the application's current memory size is not large enough to satisfy the needs, the Wasm runtime will enlarge the existing memory region. Wasm application's runtime memory is enlarged by extending the end address of the original memory's end address but keeping the same starting address. In other words, only one chunk of contiguous physical memory is allowed for each application. Wasm runtime will do a similar memory checking procedure as when instantiating an application to request Aerogel runtime to check whether such extension overlaps with memory mapped I/O addresses of the sensors or actuators, but the anticipated memory size is the application's original memory heap size plus the required enlarging memory size. If a new starting address is return by Aerogel runtime, Wasm runtime copies the contents from the old memory chunk to the new memory regions and frees the old memory trunk.

Given peripheral memory isolation, we can now explain how Aerogel's runtime enforces access control to the devices by starting from how the users should specify the access rules through an *access control specification sheet.*

```
<Device-Spec>
    <!--The metadata of the device-->
    <Param name="home_camera" description="Home smart camera"/>
    <Sub-Devices>
        <Property>
            <!--Capture the image-->
            <Param name="image_sensor" address="0x8FFFFFE0"/>
            <!--Power state, power unit mW-->
            <Max-Access val="10"/>
            <Power-State val="active" power="800"/>
            <Power-State val="sleep" power="300"/>
        </Property>
        <Property>
            <!--Adjust the angle of the camera-->
            <Param name="angle_actuator" address="0x8FFFFFF4"/>
            <Max-Access val="1"/>
            <!--Power state, power unit mW-->
            <Power-State val="active" power="1000"/>
        </Property>
    </Sub-Devices>
</Device-Spec>
```

**(a) An example of per-device specification sheet. This device specification sheet is a smart home camera that has an image sensor capturing the images and a camera angle adjustment actuator.**

```
<App-Spec>
    <Applications>
        <!--Application name and description-->
        <Param name="home_security" description="Home security App"/>

        <!--Allowed access devices and energy (uAh)-->
        <Access-Devices>
            <!--Set the consumed energy to be 0 after reset (ms)-->
            <Device name="image_sensor" energy="2500" reset="10000"/>
            <Device name="angle_actuator" energy="unlimited" reset="0"/>
            <Device name="door_actuator" energy="1500" reset="15000"/>
        </Access-Devices>

        <!--Allowed processor energy and memory usage-->
        <Processor-Memory>
            <Processor val="5000" reset="15000"/> <!--5000 uAh-->
            <Memory val="131072"/> <!--128 KB-->
        </Processor-Memory>
    </Applications>
</App-Spec>
```

**(b) An example of per-application specification sheet. This access control specification sheet is a smart home security monitoring application, which is allowed to access smart camera and door controller.**

**Figure 3: Example of the specification sheet needed by Aerogel.**

## 4.2 Access Control Specification

The Aerogel access control specification sheet defines the permission list for each application. Aerogel requires the user to provide two pieces of information through the specification sheet: 1) per-device specifications and 2) per-application specifications.
**Per-device specification.** For each device, the user needs to define the device's manufacturer information. In particular, the user should specify the power consumption profiles for each sensor, actuator, and I/O addresses as well as the processor power under different power states. The user also defines the maximum number of applications that can access each sensor or actuator at a time.
**Per-application specification.** For each application on the IoT device, the user provides an allowlist of sensors or actuators, the maximum energy usage, the maximum processor energy usage, as

well as the maximum memory usage. The user will also specify the *reset time* for resetting the application's total energy usage to be zero.

Figure 3 shows an example of the two specification sheets for Aerogel[3]. These specification sheets are provided by the user of the devices. Figure 3a shows a per-device specification sheet for a smart home security camera that has an image sensor and an angle adjustment actuator. In this example, up to ten applications can access the image sensor. The device has two power states: active and sleeping. The angle actuator is only accessible by one application at any time. Figure 3b is an example of access control per-application specification sheet. In this example, the *home_security* application is given access to the smart camera's image sensor, the angle adjustment actuator, and the door movement actuator. Access to the angle adjustment actuator allows unlimited energy usage. The total energy usage for the processor resets after 15000 ms.

Once the specification sheet policies are loaded in memory, Aerogel's runtime starts enforcing the access control rules using a hierarchy of access checkers. The first access control checker focuses on compute resource access control.

## 4.3 Compute Resource Access Control

The first stage of Aerogel's access control focuses on compute resource policies. Aerogel first checks compute resource access policies before peripheral access policies since all applications will require compute resources, but not all applications will access peripherals. Aerogel's compute access control has two components: the *memory usage checker* and the *processor energy usage checker*.

*4.3.1 Memory usage checker.* The memory usage checker performs the total memory usage checking when a new piece of memory region needs to be allocated by the Wasm runtime. Thus, this type of access control checking is triggered in two scenarios: 1) application instantiation and application runtime memory expansion. Because Wasm runtime allows only one chunk of contiguous memory for each Wasm application, the memory usage checker only needs to keep track of each application's start and end addresses when its memory regions are changed. The memory usage checker computes the total memory usage of a specific application by subtracting the application's end address from the application's start address. The difference is compared with the user-specified memory usage threshold. Hence, the performance of checking the memory usage is always constant, i.e., $O(1)$.

*4.3.2 Processor energy usage checker.* An application's processor energy usage is defined as the processor energy consumed by executing its Wasm instructions and invoked native functions. We rely on the additional counters from MCU to collect the information of the power state of the MCU at given time. Meanwhile, we use pre-profile the actual MCU instructions needed by each Wasm instruction, and we combine these two pieces of information to compute the MCU power consumption for the specific Wasm instruction. For example, suppose a Wasm instruction adds two numbers with the opcode ADDITION followed by two numbers as the arguments. In that case, the processor energy consumed is the processor energy that needs to add those two numbers, including loading them to the registers and storing the result back to the memory.

For each Wasm instruction or native function invocation, Aerogel's runtime records the total execution time under different processor states and computes the energy consumption by summing up $P_i * t_i$, where $P_1, P_2, ..., P_n$ are the different power states of the processor and $t_1, t_2, ..., t_n$ are the corresponding execution times.

After the execution of one Wasm instruction, the processor energy usage checker checks whether the application's total energy cost has exceeded the maximum allowable value. Suppose the total energy cost is more than the allowed maximum. In that case, the application is scheduled to sleep for a period defined by the user. Once the period has passed, the total energy cost resets to zero.

**Performance optimizations.**

Since one application might have many Wasm instructions, it is inefficient to compute the energy for every instruction. We introduce two optimization methods to reduce the overhead of the processor energy checking procedures.

- For the static instructions whose execution times do not change under different applications, e.g., addition and subtraction, Aerogel's runtime stores the value of their associated energy cost. When Aerogel encounters those instructions, the checker retrieves the value from the first computation.
- Similarly, when instructions have the same processor execution cycles, we only need to compute the processor energy consumption for one of them and reuse the calculated value for the rest instructions. For example, loading a 32-bit float to a register has the same number of execution cycles as loading a 32-bit integer.

## 4.4 Sensor And Actuator Access Control

Unlike the compute resource access controls checkers, the sensor and actuator access control checkers only enforce the access control rules when an application requires access to the sensors or actuators. Aerogel's sensor and actuator access control consists of three components: the *sensor and actuator initial permission checker*, the *access monitor*, and the *sensor and actuator energy usage checker*.

*4.4.1 Sensor and actuator initial permission checker.* The initial permission checker is triggered when a new application requests Aerogel's runtime to read the sensor data or write data to actuator peripherals. When such a request is received, the initial permission checker checks whether the requested sensor or actuator is in the allowlist of sensors and actuators for the application parsed from the specification sheet. If the requested sensor or actuator is on the list, the initial access checker will allow the request to advance to the next stage. Otherwise, a denial will be sent back to the application.

*4.4.2 Access monitor.* The access monitor verifies that only a certain (user-defined) number of applications are accessing the sensors and actuators, i.e., Aerogel enforces the user-defined counting semaphores for sensor and actuator peripherals. For example, for a temperature measuring application, the user only wants this application to measure the average temperature during a certain time, which can be limited by the number of allowed accesses.

[3]The grammar template can be found here: https://tinyurl.com/aerogel-spec-sheet

When an application needs to register with the sensor and actuator module, the access monitor finds the current total number of applications accessing the sensor or actuator. If the access is less than the total number of allowed accesses, the peripheral access will be granted. Aerogel's runtime then increments the total number of accesses. When an application dispatches from the sensor and actuator module, Aerogel's runtime will decrement the total number of applications accessing the sensor or actuator.

*4.4.3 Sensor and actuator energy usage checker.* When there is a new peripheral event to be handled such as new sensor data or new actuation command, the sensor and actuator module sends the power states of the sensor or actuator and the duration of that application on each power state to the sensors and actuators *energy usage checker*. The checker looks up the power information of each power state sent from the module according to the previously parsed per-device specification sheet. The checker then computes the energy usage of this event and adds this energy to the total sensor or actuator energy consumed by the application for that particular sensor or actuator.

The energy usage checker also compares whether the energy usage has exceeded the maximum allowable value. If yes, the access checker will request Aerogel's runtime to virtually dispatch the corresponding application from the sensor and actuator module, i.e., the application cannot read or write to peripherals. The application's total energy consumption for the given sensor or actuator is only reset to zero when the user-defined reset period has passed. If the application was previously dispatched from the sensors and actuators module, it would be virtually registered back with the module. All virtual dispatches or registrations do not decrease or increase the number of total accesses for that sensor or actuator.

# 5 IMPLEMENTATION

In this section, we will discuss the implementation of Aerogel.

## 5.1 System Setup

We prototyped our design Aerogel as depicted in figure 2 with the Wasm micro-runtime (WAMR) [11]–which is implemented in a mixture of C and assembly on both a bare-metal dev board (Nordic nRF52840 [43]) and a simulator (QEMU [48]). We use the Zephyr real-time OS [56] as the bare-metal operating system. The nRF52840 dev board is equipped with a 32-bit ARM Cortex-M4 MCU whose running frequency is 64 MHz with 1MB flash and 256 KB RAM. The nRF52840 is mainly utilized by wireless IoT devices such as wireless security cameras. Because we need to measure the overhead of Aerogel under different processor frequencies, we also set up the QEMU simulator with various MCU frequencies from 10 MHz to 110 MHz with 512 KB RAM.

Aerogel and the associated runtime are implemented with a total of 2321 significant Lines of Code (sLoC): 1399 sLoC for the Aerogel runtime implementation, 108 sLoC for testing and debugging purposes, and 814 sLoC for evaluation.

## 5.2 Aerogel Runtime

We now describe how we implement the three major components of the Aerogel runtime: the access control specification sheet parser, the sensor and actuator module, and the access control checkers. Moreover, we describe how we augmented our implementation to support Just-in-Time (JIT) compilation for the Wasm applications enabled by the WAMR.

*5.2.1 Access control specification sheet parser.* The parser needs to initiate two types of state variables when parsing the specification sheet: global state variables and per-application state variables. The global state variables are shared among all the applications. In particular, all variables extracted from the per-device specification sheet information are considered global variables, e.g., the address of each sensor and actuator, the power states, and the maximum allowable concurrent access to a particular sensor or actuator. The access control specification sheet initiates the global state variables only once. We implement the `parse_per_device()` function to parse the per-device access control specification sheet at the beginning of the `wasm_env_create()`–which creates the Wasm environment for all Wasm applications.

Per-application state variables are parsed from the per-application specification sheets and vary for different applications. We implement the `parse_per_app()` function to parse the per-application access control specification sheet such that each application's variables are initialized. These variables include information about the allowable set of sensors and actuators, the energy allowed, and the associated reset time. The access control checker will use the variables after each application is instantiated by the `wasm_instantiate()` function–which initiates the Wasm application runtime for a particular application.

*5.2.2 Sensor and Actuator Module.* We implement the sensor and actuator module as native functions that are exported and exposed to the Wasm applications. When a Wasm application calls the sensor and actuator module functions, the Wasm runtime looks into a function table pre-registered with all native functions and finds the symbols of the sensor and actuator module functions. The symbols are linked with the Wasm application at runtime.

We implement two APIs for application developers, as summarized in table 2. The sensing API is used to register the application to listen to any sensing events, and the actuation API is used to send an actuation command to the actuators from the application. When a Wasm application invokes either of these APIs, the sensing and actuating functions will first call the `access_control_checking()` function of the Aerogel runtime to ensure such a request is legitimate. If the request is to periodically send actuation commands or receive sensing data, the sensing or actuating functions will call the energy and memory usage checking functions to ensure the total energy and memory usage has not exceeded the application's allotment.

*5.2.3 Access Control Checkers.* The sensor and actuator access checkers' implementation is integrated with the sensor and actuator module. In particular, the sensor and actuator initial permission checker and the access monitor are called at the beginning of the functions `sensing (id, freq, duration, cb_func)` and `actuation(id, *value, cb_func)` before executing the sensing or actuation requests. The sensor and actuator energy usage checker is implemented at the sensor and actuator module before the peripheral request is executed. The reset timer used by the

**Table 1: Simulated sensors and actuators[4] for Unmanned Aerial Vehicles (UAVs) and smart home for Aerogel evaluations.**

| Category | Device Name | Peripherals | Description | Index | Power | Max Concurrent Access |
|---|---|---|---|---|---|---|
| Unman. Aerial Vehicle (UAV) | Camera | Camera image sensor | Capturing the images | ① | 2W | 5 |
| | GPS | GPS sensor | Sensing GPS positioning signals | ② | 1W | 10 |
| | Propellers | Motor actuators | Controlling UAV propeller motors | ③ | 10W | 1 |
| Smart Home | Camera | Camera image sensor | Capturing images | ④ | 5W | 5 |
| | | Angle control actuator | Controlling camera angles | ⑤ | 4W | 1 |
| | Door Control | Motion Sensor | Detection moving objects | ⑥ | 0.2W | 10 |
| | | Door motor actuator | Controlling door opening and closing | ⑦ | 3W | 1 |
| | | Battery usage sensor | Detecting battery capacity | ⑧ | 0.2W | 10 |
| | Speaker | Speaker actuator | Playing sound from the speaker | ⑨ | 4W | 1 |
| | Microphone | Microphone sensor | Sensing acoustic signals | ⑩ | 1W | 2 |

**Table 2: APIs implemented of the sensor and actuator module exposed to the Wasm application developer.**

| APIs | Description |
|---|---|
| `sensing (id, freq, duration, cb_func)` | Register the application to the sensor with *id* for given *frequency* and *duration*. The results are sent back by the *callback* function. |
| `actuation(id, *value, cb_func)` | Send the actuation command with the expected *value* to the actuator with *id*. The returned values if any from the actuator are sent back through the *callback* function. |

Aerogel access control checkers to reset the total peripheral energy is realized with the Zephyr up-time timer, which is the time relative to the board's boot-up time.

The compute resource access checkers are implemented where a Wasm native function or bytecode instruction is called. For the memory usage checker, it is invoked when extra memory is needed for the Wasm application's runtime. In particular, the memory usage checker is implemented at the beginning of `wasm_instantiate()` function that instantiates the Wasm runtime environment for the application and the `wasm_enlarge_memory()` function that requests extra memory when the current memory is not large enough. The processor energy usage checker is implemented at the end of the execution of each Wasm bytecode instruction or native function.

# 6 EVALUATION

We evaluate our design Aerogel on both nRF5840 dev board and QEMU. We first explain the benchmarks we used for our evaluation, followed by the experimental results.

## 6.1 Benchmarks

To evaluate our design, we first implemented several simulated sensors and actuators for Unmanned Aerial Vehicles(UAVs) and smart home environments. For the UAVs, we simulated a camera, a GPS, and the motor for the propellers. For the smart home scenario, we simulated four different devices that have more than one sensor or actuator, e.g., a smart home camera and a door controller. Table 1 summarizes all of the simulated devices.

To the best of our knowledge, our work is the first known to use Wasm for bare-metal device access control. We proposed micro-benchmarks based on real-life examples of UAV systems and smart home systems. The sensing or actuation rate and execution time we chose in the benchmark is to closely imitate the real sensing or actuation rate people use for their sensors and actuators in their life. We evaluated eight different Wasm sensing and actuation applications, as summarized in table 3 based on the sensors and actuators of the UAVs and the smart home. Among these eight Wasm sensing and actuation applications, four of them have regular access to the sensors or actuators. The other four are restricted to evaluate denial for certain access requests.

The regular access Wasm applications for UAVs are the *uav_ctrl* that is designed to be the UAV flight control system, and *uav_sense* that is used to capture an image through the camera of the UAV. In the smart home scenario, we proposed a *home_monitor* application that monitors the home status through the available sensors and a *home_security* application that protects the safety of the home. The four restricted access applications are used to evaluate the four different access control checkers under extreme conditions such as a shortage of processor energy consumption and sensor energy consumption, maximum concurrent accesses to peripherals, and initial access to peripherals denial. For each sensor or actuator the applications try to access, we set the duration for one second.

## 6.2 Results

We analyze the results of the benchmarks on the nRF52840 board and QEMU. We combined the overhead of the initial access checker, the memory usage checker, and the maximum concurrent access checker for all of our results. The total overhead in the worst case is less than 0.07% of the execution time.

*6.2.1 Latency overhead.* We run all benchmark Wasm applications on the nRF52840 board, as summarized in table 4. The reported overhead in this table does not include the specification sheet parsing since this is done only at the system boot up time, where the average overhead of parsing the specification sheet is 450ms **that occupies 30% of the overall system bootup time**.

Our results show that for the regular access applications, Aerogel runtime introduces at most 1.04% overhead. Most of the overhead comes from the sensor and the actuator energy checker, whose energy checking happens more frequently than the other checkers. For the UAV sensing application, the overhead is only 0.19% of the total execution. This reduction comes from the shorter execution time of sensing and actuating energy consumption checkers. Given that there are more Wasm instructions in UAV control applications and the Aerogel access control checkers for each Wasm application consume more cycles than energy checkers, we expect such

**Table 3: Benchmark applications running on nRF52840 board and its access configurations.**

| | App Name | Description | Devices Used | # of Wasm inst.[1] | Wasm file size (Bytes) | Allowed Devices | Device Energy Allowed | MCU Energy Allowed | Memory Usage Allowed |
|---|---|---|---|---|---|---|---|---|---|
| Regular Access | uav_ctrl | UAV flight control system | ①②③ | 75 | 542 | ①②③ | 100 mAh | 20 µAh | Unlimited |
| | uav_sense | UAV image capturing based on different locations | ①② | 41 | 362 | ①② | 50 mAh | Unlimited | 250 KB |
| | home_monitor | Voice control to get home info and play it via the speaker | ④⑧⑨⑩ | 86 | 607 | ④⑧⑨⑩ | Unlimited | Unlimited | 255 KB |
| | home_security | Door opening after image identity verification | ④⑤⑥⑦ | 70 | 565 | ④⑤⑥⑦ | 100 mAh | Unlimited | 230 KB |
| Restrict. Access | uav_shortage_mcu | Exceeds max allowed MCU power usage on UAV | ① | 41 | 374 | ① | 10 mAh | **0.5 µAh** | 250 KB |
| | home_shortage_cam | Exceeds max allowed home camera power usage | ④ | 41 | 393 | ④ | **1.5 µAh** | 60 µAh | 240 KB |
| | uav_max_access | Exceeds max allowed access to UAV propellers | ①③ | 77 | 549 | ①③ | 60 mAh | 70 µAh | 250 KB |
| | home_init_denial | Access to some smart home sensors denied | ④⑧⑩ | 55 | 490 | ④ | 60 mAh | 50 µAh | 200 KB |

**Table 4: Aerogel overhead of benchmark applications running on nRF52840 dev board.**

| Aerogel Overhead Breakdown | | | | | | |
|---|---|---|---|---|---|---|
| Percentage over Total time | Processor Energy Checker | Sen./Act. Energy Checker | Other Checkers | **Total Aerogel Overhead** | App Execution | Total time (ms) |
| uav_ctrl | 0.01% | 1.02% | 0.01% | 1.04% | 98.96% | 3166 |
| uav_sense | 0.00% | 0.17% | 0.01% | 0.19% | 99.81% | 1056 |
| home_monitor | 0.00% | 0.92% | 0.02% | 0.94% | 99.06% | 4127 |
| home_security | 0.00% | 0.58% | 0.02% | 0.61% | 99.39% | 4092 |
| uav_shortage_mcu | 0.01% | 0.12% | 0.01% | 0.14% | 99.86% | 1530 |
| uav_shortage_cam | 0.01% | 0.52% | 0.02% | 0.55% | 99.45% | 1171 |
| uav_max_access | 0.01% | 0.17% | 0.06% | 0.24% | 99.76% | 1064 |
| home_init_denial | 0.01% | 0.09% | 0.07% | 0.16% | 99.84% | 1054 |

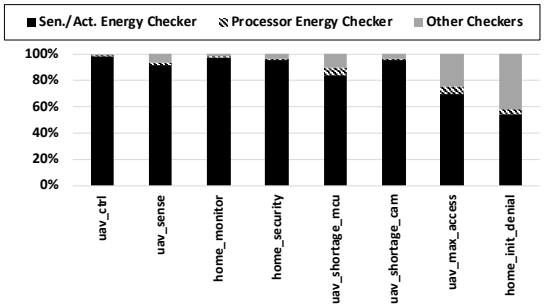


**Figure 4: The overhead percentage of different Aerogel runtime access control checkers under each Wasm application on nRF52840 board.**

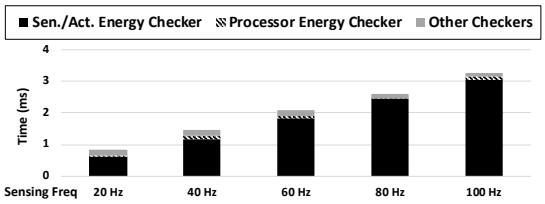


**Figure 5: The overhead of Aerogel for *uav_sense* on nRF52840 board under various camera sensing frequency.**

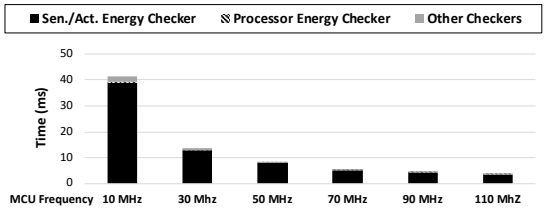


**Figure 6: The overhead of Aerogel for *home_security* on QEMU simulator under various MCU frequency.**

reduction. When examining the runtime overhead of the restricted access applications, we found the sensor energy shortage application has the most overhead at 0.55%. This overhead is the result of the applications that require frequent access to sensors and, thus, more energy checks when new sensor data is available. On the other hand, the lowest MCU application's overhead is 0.14%. When the application's energy usage is denied, the application is scheduled to sleep immediately–resulting in fewer checks than other applications.

We examined the overhead percentage of different access checks relative to the total Aerogel overhead. The results–depicted in Figure 4–show that the sensor and the actuator energy usage checker consumes the most overhead. The energy usage checker is triggered when a new sensor event or actuation command needs to be handled. Some applications, e.g., the *uav_control* that sends more than 1000 actuation commands, trigger thousands of sensor and energy usage checking procedures. In contrast, the processor energy usage checker triggers only tens of times, as shown in table 3. Hence, the overhead of the sensor and actuator energy usage checker is significantly higher than that of the processor energy usage checker. Other access control checkers, such as the initial access control checker, consume a large portion of the overhead only in restricted access applications, especially the initial access denial app. This overhead is due to the high-frequency sensor's access denial, resulting in fewer energy usage checks.

Moreover, the overhead introduced by the checkers of Aerogel is likely correlated to the access frequency to the required sensors and actuators. For example, in the *uav_ctrl* application, the sensors and actuators are more frequently accessed than other applications. Hence, the overall overhead is contributed most from the sensor and actuator energy checker.

We ran the experiment under different sensing frequencies on the nRF52840 dev board and different processor frequencies on the QEMU emulator. Figure 5 shows the different camera sensing

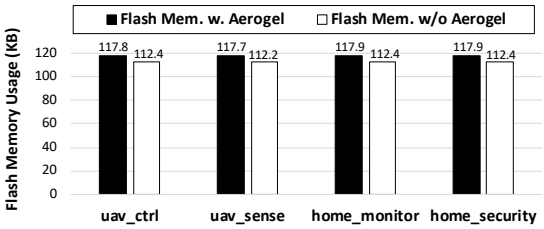


**Figure 7: The flash memory usage of nRF52840 board when running different applications.**

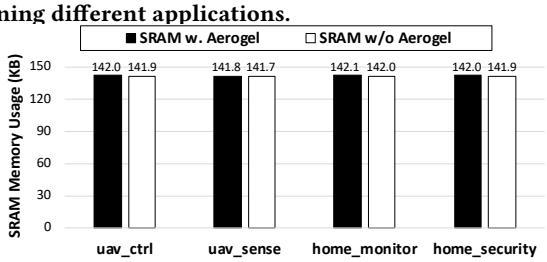


**Figure 8: The SRAM usage of nRF52840 board when running different applications.**

frequencies of the *uav_sense* application. The overhead of the sensor and actuator energy usage checker increases with respect to the increase of the sensing frequency. In contrast, the overhead of other access control checker does not increase. The sensing frequency change only increases the number of sensing events that trigger the sensor energy usage checking. We also show the overhead of Aerogel runtime under different MCU frequencies on the QEMU simulator, as illustrated in Figure 6. Aerogel runtime has very minimal overhead when the MCU frequency is greater than or equal to 70 MHz.

*6.2.2 Memory overhead.* We next evaluated Aerogel's SRAM and flash memory overhead. We only evaluated the memory overhead for the four regular access applications since the restricted access applications cannot provide full execution paths. The flash memory size is the size of all the compiled code–including the OS, the applications, and the Aerogel runtime. For the flash memory, Aerogel runtime only introduces a marginal overhead that is less than 5KB–independent of the application. The Aerogel runtime overhead of SRAM is also minimal around 0.1KB. This minimal overhead is due to the fact that no significant amount of temporary data is stored in memory.

*6.2.3 Energy consumption.* Finally, we evaluated nRF52840's board energy consumption for our benchmark. To measure the energy consumption, we used the Monsoon power monitor [41] to connect to the dev board. The monitor provides a 3V external battery for the board. Our results show that the Aerogel runtime costs a maximum of 0.65 $\mu Ah$ and a minimum of 0.39 $\mu Ah$ for all regular access applications. For the restricted access control applications, the energy consumption with Aerogel runtime enabled for some applications such as *home_init_denial* and *uav_max_access* is less than those without enabling Aerogel runtime. This is because

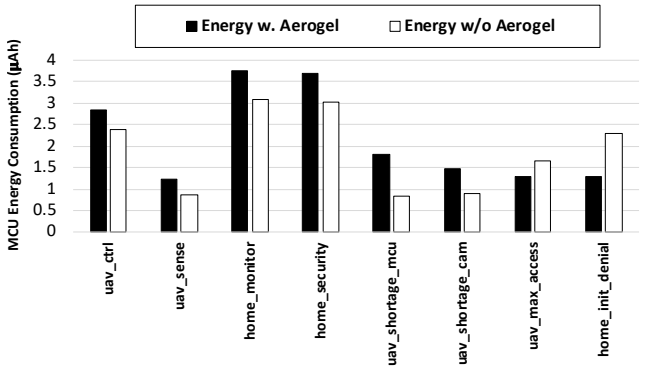


**Figure 9: The energy consumption of nRF52840 board when running different applications.**

when such access is denied, Aerogel runtime does not need to have further energy usage check of the sensor or actuators. The application is not allowed to further execute to read the sensor data or to send the actuation commands. It is the application developers' responsibility to take care of such access denials, otherwise, a rescue control will be triggered to take over the control of the device. The reason why the relative energy consumption is higher than the relative overhead is that the energy checking is at high energy consumption mode because the MCU is always awake and executing, but the benchmark applications are I/O intensive applications.

## 7 DISCUSSION AND FUTURE WORK

We discuss the security analysis and future work of Aerogel.

**Attack prevention.** Due to the memory protection characteristics of Wasm, Aerogel can build a secure runtime that provides software memory isolation for all peripherals even when the devices lack the memory address space separation from MMU. Aerogel also protects the bare-metal IoT devices from malicious applications that try to drain the device's resources such as the battery and memory. Moreover, Aerogel can protect the sensors and actuators of the device from being accessed without the user's authorizations. Further, Aerogel's processor energy usage checker can mitigate a Denial-of-Service (DoS) attack from a malicious Wasm application because it has a restricted energy and memory usage allotment.

**Limitations.** Aerogel cannot protect against side-channel memory attacks [36]. Aerogel also cannot protect against cyber-physical attacks on the peripherals such as GPS spoofing. Furthermore, Aerogel cannot prevent the application from misusing the sensor data or sending dangerous actuation commands if it is granted access to the sensor or the actuator. Finally, Aerogel cannot provide access control for sensors or actuators that are not memory-mapped I/Os, e.g., interrupt-based sensors, because Wasm runtime only provides the isolation of the memory although most sensors on SoC nowadays are memory-mapped sensor. If we would further like to support the protections for non-memory-mapped I/Os as well, we will need the extra hardware protections such as the Memory Protection Unit (MPU) in Arm-M series to separate the main memory bus and the interrupt devices.

**Future work.** Our evaluation did not explore the adaptation of Aerogel to dynamic instantiation of application workloads across IoT devices. In the context of dispersed IoT computational networks where devices may leverage opportunistically available resources [28, 42], Aerogel would need to adapt to applications that are dynamically streamed from other devices. In this context, the specification and enforcement of access control policies would need to be dynamic and partially autonomous. Future directions can also explore optimization of access control enforcement in the face of third-party, dynamically streamed and instantiated application workloads. Although Wasm's runtime (e.g., Wamr) supports dynamic application installation, future work can minimize the impact of executing applications when the over-the-air installation is happening given the limited resources on bare metal IoT devices.

## 8 RELATED WORK

We now compare the existing work with Aerogel.

**Memory Protection on Bare-metal Devices.** ACES [13], MicroGuard [49], uXOM [35] and Clements et al. [14] use the compiler to achieve the memory compartmentalizing on bare-metal devices based on ARM Cortex-M's Memory Protection Unit (MPU). PicoXom [52] uses MPU to provide read-only memory execution environment on ARM Cortex M environment. $\mu$RAI [3] enforces Return Address Integrity (RAI) by modifying the compiler to move the return address from writable memory to read- and execute-only memory regions. Moreover, ARMor [57] uses formal verification methods of software fault isolation (SFI) to ensure the memory safety and the control flow integrity of applications by inserting dynamic check before certain instructions. Unlike them, Aerogel is able to achieve memory isolation including the protections to the peripherals at runtime without pre-inspecting the applications. Prior works [17, 37, 46] use MPU to achieve runtime memory protection. However, the applications need to be rewritten under different bare-metal OSes, while those applications on Aerogel do not need to be reprogrammed for different bare-metal OSes. Some distributed lower-level protections [15, 20, 32] such as the firmware-level can provide the protections for specific devices. Aerogel provides access protection for general IoT devices that support the Wasm execution environment.

**Access Control on IoT Devices.** AccTee [21] uses Intel SGX with Wasm to enforce access control usage for the memory and CPU usage for cloud applications. Several prior works [4, 29, 40, 45, 47] adopt blockchain techniques to achieve decentralized access control for IoT devices. Atlam et al. [8] builds an access control model based on the context of the environment to decide whether granting the access request exposes the security risks of the data usage. However, although the above works propose access control framework for IoT networks, these works fail to provide fine-grained access control for applications running on an individual device as in Aerogel.

**Secure Microkernels.** Secure-formal verification-based microkernels such as SeL4 [33] are also capable providing fine-grained access control under certain circumstances. However, the microkernels lack easily fine-grained control of instruction execution, while Wasm JIT runtime can provide an accurate measurement of the MCU time for a given Wasm instruction, which makes microkernels hard to control the processor power consumption that can be consumed by different applications. Moreover, unlike Aerogel, SeL4 mainly targets on the powerful processors such as ARM A-series platforms due to the performance and design restrictions (e.g. lack of hardware assisted memory protections) of the micro-controller platforms such as ARM M-series SoCs [27].

**Wasm on the Edge.** OneOS [31] designs a single-image universal edge OS for heterogeneous IoT devices using JavaScript that can enable Wasm execution. Hall et al. [26] utilizes Wasm to execute serverless functions on edge to reduce the hardware resources usage with respect to traditional edge serverless computation systems. Wasmachine [54] uses Wasm to host an edge operating system with kernel written by Rust to speed up the applications running on top of it. Jeong et al. [30] proposes a system offloading Wasm functions mingled with JavaScript to edge server from the mobile devices to reduce execution latency. However, unlike Aerogel, none of the above focuses on access protections to use peripherals such as sensors and actuators on bare-metal IoT devices where MMU is not available.

## 9 CONCLUSION

In this paper, we propose Aerogel, a lightweight access control framework to define fine-grained access control policies for Wasm-based, bare-metal IoT devices. Aerogel leverages the security features of Wasm runtime to protect the access and usage of peripherals. We prototype Aerogel on nRF52840 dev board, and the results show that Aerogel only introduces 0.19% to 1.04% overhead.

## ACKNOWLEDGMENTS

The research reported in this paper was sponsored in part by the National Science Foundation (NSF) under award #CNS-1705135, by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, and by the Army Research Laboratory (ARL) under Cooperative Agreement W911NF-17-2-0196. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARL, DARPA, NSF, SRC, or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation here on.

## REFERENCES

[1] https://medium.com/@torch2424/webassembly-is-fast-a-real-world-benchmark-of-webassembly-vs-es6-d85a23f8e193.
[2] Alhadlaq, A., Tang, J., Almaymoni, M., and Korolova, A. Privacy in the amazon alexa skills ecosystem. *Star 217*, 11 (1902).
[3] Almakhdhub, N. S., Clements, A. A., Bagchi, S., and Payer, M. $\mu$rai: Securing embedded systems with return address integrity.
[4] Andersen, M. P., Kumar, S., AbdelBaky, M., Fierro, G., Kolb, J., Kim, H.-S., Culler, D. E., and Popa, R. A. WAVE: A decentralized authorization framework with transitive delegation. In *28th USENIX Security Symposium (USENIX Security 19)* (Santa Clara, CA, Aug. 2019), USENIX Association, pp. 1375–1392.
[5] ARM. Arm cortex m4. https://developer.arm.com/ip-products/processors/cortex-m/cortex-m4.
[6] ARM. Arm cortex m7. https://developer.arm.com/ip-products/processors/cortex-m/cortex-m7.
[7] ARM. The memory management unit (mmu). https://developer.arm.com/architectures/learn-the-architecture/memory-management/the-memory-management-unit-mmu.

[8] Atlam, H. F., Alenezi, A., Walters, R. J., Wills, G. B., and Daniel, J. Developing an adaptive risk-based access control model for the internet of things. In *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)* (2017), pp. 655–661.

[9] Avramidis, I., Mackay, M., Tso, F. P., Fukai, T., and Shinagawa, T. Live migration on arm-based micro-datacentres. In *2018 15th IEEE Annual Consumer Communications & Networking Conference (CCNC)* (2018), IEEE, pp. 1–6.

[10] Bai, Y. Arm® memory protection unit (mpu).

[11] BytecodeAlliance. Wasm micro-runtime. https://github.com/bytecodealliance/wasm-micro-runtime.

[12] Clark, L. Announcing the bytecode alliance: Building a secure by default, composable future for webassembly. https://bytecodealliance.org/articles/announcing-the-bytecode-alliance, 2020. Last accessed: 2020-07-29.

[13] Clements, A. A., Almakhdhub, N. S., Bagchi, S., and Payer, M. ACES: Automatic compartments for embedded systems. In *27th USENIX Security Symposium (USENIX Security 18)* (Baltimore, MD, 2018), USENIX Association, pp. 65–82.

[14] Clements, A. A., Almakhdhub, N. S., Saab, K. S., Srivastava, P., Koo, J., Bagchi, S., and Payer, M. Protecting bare-metal embedded systems with privilege overlays. In *2017 IEEE Symposium on Security and Privacy (SP)* (2017), pp. 289–303.

[15] Clements, A. A., Gustafson, E., Scharnowski, T., Grosen, P., Fritz, D., Kruegel, C., Vigna, G., Bagchi, S., and Payer, M. Halucinator: Firmware rehosting through abstraction layer emulation. In *29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), USENIX Association, pp. 1201–1218.

[16] Community, W. Webassembly. https://webassembly.org/.

[17] Danner, D., Müller, R., Schröder-Preikschat, W., Hofer, W., and Lohmann, D. Safer sloth: Efficient, hardware-tailored memory protection. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2014), pp. 37–48.

[18] Dronecode. Pixhawk 4. https://docs.px4.io/v1.9.0/en/flight_controller/pixhawk4.html.

[19] Ensergueix, T. Cortex-m23 and cortex-m33 - security foundation for billions of devices, October 2016.

[20] Feng, B., Mera, A., and Lu, L. P2im: Scalable and hardware-independent firmware testing via automatic peripheral interface modeling. In *29th USENIX Security Symposium (USENIX Security 20)* (Aug. 2020), USENIX Association, pp. 1237–1254.

[21] Goltzsche, D., Nieke, M., Knauth, T., and Kapitza, R. Acctee: A webassembly-based two-way sandbox for trusted resource accounting. In *Proceedings of the 20th International Middleware Conference* (New York, NY, USA, 2019), Middleware '19, Association for Computing Machinery, p. 123–135.

[22] Google. Nest protect. https://store.google.com/us/product/nest_protect_2nd_gen.

[23] Gudeth, K., Pirretti, M., Hoeper, K., and Buskey, R. Delivering secure applications on commercial mobile devices: The case for bare metal hypervisors. In *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices* (New York, NY, USA, 2011), SPSM '11, Association for Computing Machinery, p. 33–38.

[24] Haas, A., Rossberg, A., Schuff, D. L., Titzer, B. L., Holman, M., Gohman, D., Wagner, L., Zakai, A., and Bastien, J. Bringing the web up to speed with webassembly. *SIGPLAN Not. 52*, 6 (June 2017), 185–200.

[25] Habibzadeh, H., Qin, Z., Soyata, T., and Kantarci, B. Large-scale distributed dedicated-and non-dedicated smart city sensing systems. *IEEE Sensors Journal 17*, 23 (2017), 7649–7658.

[26] Hall, A., and Ramachandran, U. An execution model for serverless functions at the edge. In *Proceedings of the International Conference on Internet of Things Design and Implementation* (New York, NY, USA, 2019), IoTDI '19, Association for Computing Machinery, p. 225–236.

[27] Heiser, G. The sel microkernel an introduction.

[28] Hu, D., and Krishnamachari, B. Throughput optimized scheduler for dispersed computing systems. In *2019 7th IEEE International Conference on Mobile Cloud Computing, Services, and Engineering (MobileCloud)* (2019), IEEE, pp. 76–84.

[29] Hwang, D., Choi, J., and Kim, K. Dynamic access control scheme for iot devices using blockchain. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)* (2018), pp. 713–715.

[30] Jeong, H.-J., Shin, C. H., Shin, K. Y., Lee, H.-J., and Moon, S.-M. Seamless offloading of web app computations from mobile device to edge clouds via html5 web worker migration. In *Proceedings of the ACM Symposium on Cloud Computing* (New York, NY, USA, 2019), SoCC '19, Association for Computing Machinery, p. 38–49.

[31] Jung, K., Gascon-Samson, J., and Pattabiraman, K. Oneos: Iot platform based on POSIX and actors. In *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)* (Renton, WA, July 2019), USENIX Association.

[32] Kim, C. H., Kim, T., Choi, H., Gu, Z., Lee, B., Zhang, X., and Xu, D. Securing real-time microcontroller systems through customized memory view switching. In *NDSS* (2018).

[33] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. Sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (New York, NY, USA, 2009), SOSP '09, Association for Computing Machinery, p. 207–220.

[34] Klingensmith, N., and Banerjee, S. Hermes: A real time hypervisor for mobile and iot systems. In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications* (2018), pp. 101–106.

[35] Kwon, D., Shin, J., Kim, G., Lee, B., Cho, Y., and Paek, Y. uxom: Efficient execute-only memory on ARM cortex-m. In *28th USENIX Security Symposium (USENIX Security 19)* (Santa Clara, CA, Aug. 2019), USENIX Association, pp. 231–247.

[36] Lehmann, D., Kinder, J., and Pradel, M. Everything old is new again: Binary security of webassembly. In *29th {USENIX} Security Symposium ({USENIX} Security 20)* (2020), pp. 217–234.

[37] Levy, A., Campbell, B., Ghena, B., Giffin, D. B., Pannuto, P., Dutta, P., and Levis, P. Multiprogramming a 64kb computer safely and efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles* (New York, NY, USA, 2017), SOSP '17, Association for Computing Machinery, p. 234–251.

[38] Liu, R., and Srivastava, M. *VirtSense: Virtualize Sensing through ARM TrustZone on Internet-of-Things.* Association for Computing Machinery, New York, NY, USA, 2018, p. 2–7.

[39] Liu, R., Wang, Z., Garcia, L., and Srivastava, M. Remediot: Remedial actions for internet-of-things conflicts. In *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation* (New York, NY, USA, 2019), BuildSys '19, Association for Computing Machinery, p. 101–110.

[40] Lunardi, R. C., Michelin, R. A., Neu, C. V., and Zorzo, A. F. Distributed access control on iot ledger-based architecture. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium* (2018), pp. 1–7.

[41] Monsoon Solutions Inc. https://www.msoon.com/online-store.

[42] Noor, J., Tseng, H.-Y., Garcia, L., and Srivastava, M. Ddflow: visualized declarative programming for heterogeneous iot networks. In *Proceedings of the International Conference on Internet of Things Design and Implementation* (2019), ACM, pp. 172–177.

[43] Nordic Semiconductor. https://www.nordicsemi.com/Products/Low-power-short-range-wireless/nRF52840.

[44] NXP. Nxp edgeready. https://www.nxp.com/applications/solutions/enabling-technologies/edgeverse/edgeready:EDGEREADY.

[45] Ourad, A. Z., Belgacem, B., and Salah, K. Using blockchain for iot access control and authentication management. In *Internet of Things – ICIOT 2018* (Cham, 2018), D. Georgakopoulos and L.-J. Zhang, Eds., Springer International Publishing, pp. 150–164.

[46] Pan, R., Peach, G., Ren, Y., and Parmer, G. Predictable virtualization on memory protection unit-based microcontrollers. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)* (2018), pp. 62–74.

[47] Putra, G. D., Dedeoglu, V., Kanhere, S. S., and Jurdak, R. Trust management in decentralized iot access control system, 2019.

[48] QEMU. https://www.qemu.org/.

[49] Salehi, M., Hughes, D., and Crispo, B. Microguard: Securing bare-metal microcontrollers against code-reuse attacks. In *2019 IEEE Conference on Dependable and Secure Computing (DSC)* (2019), pp. 1–8.

[50] Samie, F., Tsoutsouras, V., Bauer, L., Xydis, S., Soudris, D., and Henkel, J. Computation offloading and resource allocation for low-power iot edge devices. In *2016 IEEE 3rd World Forum on Internet of Things (WF-IoT)* (2016), IEEE, pp. 7–12.

[51] Security, B. H. Blink xt2. https://blinkforhome.com/products.

[52] Shen, Z., and Criswell, J. Fast execute-only memory for embedded systems, 2020.

[53] STL. 5g and edge computing: Why does 5g needs edge compute?, Jun 2020.

[54] Wen, E., and Weber, G. Wasmachine: Bring iot up to speed with a webassembly os. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)* (2020), pp. 1–4.

[55] Yun, M. H., and Zhong, L. Securing real-time microcontroller systems through customized memory view switching. In *22nd Network and Distributed Security Symposium (NDSS 2018)* (San Diego, CA, 2018).

[56] Zephyr. https://www.zephyrproject.org/.

[57] Zhao, L., Li, G., De Sutter, B., and Regehr, J. Armor: Fully verified software fault isolation. In *Proceedings of the Ninth ACM International Conference on Embedded Software* (New York, NY, USA, 2011), EMSOFT '11, Association for Computing Machinery, p. 289–298.