University of California
Santa Barbara

A Special Purpose Operating System for Multiscale IoT Microservices

UCSB Technical Report 2022-03

A dissertation submitted in partial satisfaction
of the requirements for the degree

Doctor of Philosophy
in
Computer Science

by

Mehmet Fatih Bakir

Committee in charge:

Professor Rich Wolski, Chair
Professor Amr El Abbadi
Professor Chandra Krintz

May 2022

The Dissertation of Mehmet Fatih Bakir is approved.

Professor Amr El Abbadi

Professor Chandra Krintz, Committee Co-Chair

Professor Rich Wolski, Committee Co-Chair

May 2022

A Special Purpose Operating System for Multiscale IoT Microservices

UCSB Technical Report 2022-03

by

Mehmet Fatih Bakir

For my parents, my brother and sister in law, our cats, and Göksu who had to endure the microcontroller and cable mess over the years.

Acknowledgements

I would like to thank my advisors Chandra Krintz and Rich Wolski who supported and guided my academic journey, and supported me not only academically, but also personally in any way they could. I cannot imagine anybody else agreeing to do operating systems research in the 21st century.

I am grateful to Amr El Abbadi for joining my committee and offering insights for improving my work. The classes I took from him many years still have a profound impact on my research and work.

I want to thank Ahmet Oğuz Akyüz, who put me on the path to join the PhD program in the last minute, without whom this thesis would literally not exist.

I am extremely grateful to my wife who shared this journey, and with her unending support, made every challenge a bit less stressful.

Finally, I want to thank my family, for all the support and love over the years, even when the pandemic separated us.

Note that this thesis contains text and figures from prior publications of the author.

Abstract

A Special Purpose Operating System for Multiscale IoT Microservices

UCSB Technical Report 2022-03

by

Mehmet Fatih Bakir

The Internet of Things (IoT) enables scalable and programmable physical sensing and actuation, however many of the computers involved in IoT deployments have a wide range of capability and resource constraints. For this reason, embedded systems technologies and practices have become the predominant programming model for IoT. However, IoT also relies on scalable technologies, such as cloud computing, to provide the resources (compute, networking, and storage) required by applications. The embedded systems programming model and technology ecosystem shares little with the models and approaches typically available in cloud computing settings.

In this dissertation, we explore new research that unifies embedded systems and cloud programming, deployment, and execution for IoT applications. Our approach is based on microservices – an architecture originally formulated to ease the development of cloud-based web services. We discuss how we can extend the microservice model to resource constrained microcontrollers so that an IoT application amalgamating devices and cloud resources consists only of portable software service components that use a common set of abstractions. We describe new abstractions, how they can be secured from attackers and eavesdroppers efficiently, and the design of an operating system supporting these abstractions to unify cloud servers and microcontrollers. We evaluate and demonstrate the flexibility, portability, and efficiency of our approach using end-to-end IoT applications and microbenchmarks. Our evaluation shows that this unique combination of advances

achieves high performance and efficiency across all IoT tiers (sensors, network edge, and cloud) for the devices, systems, benchmarks, and applications that we study.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The Internet of Things (IoT) embeds ordinary objects in our environment with *digital intelligence* – via sensing, control, communications, and compute capabilities. By making it easy and economical to extract inferences and predictions from any physical object and location, IoT has the potential for unparalleled societal impact. However, IoT application development is nascent despite its tremendous potential (e.g. enabling "smart" cities, cars, homes, infrastructure, manufacturing, agriculture, etc.). The primary reason for this is that IoT software is challenging to develop, deploy, and maintain, while ensuring correctness, security, and fault resiliency required by IoT settings.

Beyond the traditional distributed systems challenges, IoT applications developers face additional difficulties that are imposed by the vast heterogeneity of IoT deployments. IoT deployments span multiple computational (i.e. resource) tiers. Devices embedded in and distributed across our physical surroundings are commonly referred to as being located at the "edge" of the network. Edge devices include a wide range of sensors, networks, compute, and storage devices, many of which are battery powered and severely resource constrained (e.g. microcontrollers and single board computers with tens of MHz processor clocks, dozens of kilobytes of main memory, and hundreds of kilobytes

of program memory). Resource constrained devices commonly offload computation and communication (to extend their capabilities and to reduce response latency) to more capable, line powered, and less resource constrained edge systems, which include small data centers, cloudlets [176], and multiprocessor servers. Edge systems inter-operate with remote cloud computing systems when more resources, services, and capabilities are needed. However, connectivity to cloud systems can be intermittently available, high latency, low bandwidth, and costly (which must be accounted for in application deployment designs).

The software stacks of these multi-scale devices also vary widely. IoT applications today run over multiple, incompatible operating systems (1+ for the embedded systems and 1+ for more resources rich systems) and communicate using different network protocols (private IP networks, low-power networks such as XBee, as well as point-to-point links such as USB, SPI and UART). Commodity operating systems for resource rich systems trade off resource efficiency for runtime generality. While Linux can run any application, it cannot run any specific application optimally, nor does it support severely resource restricted or special purpose devices. On the other hand, more specialized operating systems for embedded systems sacrifice programmer productivity and hardware abstraction for absolute runtime efficiency. A FreeRTOS [70] or bare metal application can take full advantage of a specific microcontroller, at the expense of productivity and portability. No IoT programming system today makes it possible to execute the *same code* across IoT tiers (e.g., device-edge-cloud) which severely limits code reuse, introduces bugs and vulnerabilities, and complicates inter-operation across devices.

Moreover, no extant IoT system offers a way to externally specify security properties or to perform configuration validation, precluding early detection of security vulnerabilities or the execution of a deployment plan for security mechanisms that takes into account the heterogeneous nature of IoT devices. Many edge devices have simple

(or non-existent) security mechanisms that cannot be used as-is to implement secure communication, authentication, and authorization. Existing edge solutions compound the problem via incompatible security protocols, tool chains, software development kits (SDKs) language frameworks, interfaces, and complex configuration and deployment processes. Such disparity and lack of programming, security, and deployment support makes IoT infrastructure costly to provision and maintain and precludes all but the most expert developers from contributing to IoT innovation.

Many of these productivity-limiting challenges do not exist in cloud-only programming systems. A key reason for this is the emergence of and wide spread use of the "microservice" software architectural design. Using this architecture, applications developers decompose their applications into fine-grained, event driven services that execute independently and in isolation (typically using Linux containers or other forms of lightweight virtualization). The microservices communicate via typed interprocess communication (IPC) channels with other microservices that are either co-located on the same machine or available across a network. Microservice applications do not interact directly with the operating system in the same way as traditional applications (via POSIX system calls) – instead they request/receive the resources they need, e.g. storage, from other services using well defined (typed) application programming interfaces (APIs). The lack of strong coupling between microservices and the operating system facilitates portability, rapid development, and low maintenance. Moreover, this design enables horizontal scaling with little involvement from programmers, since a particular dependency of a service can be transparently replicated. Examples of microservice platforms include Kubernetes [112], AWS Container Service [163], AWS Lambda [22], and a number of programming language frameworks [16, 14, 15]. AWS Lambda is interesting (and similar to Google Functions [76] and Azure Functions [24]) in that it specializes the microservices architecture with platform level automation (e.g. auto-scaling, event triggering, logging) and abstraction of

the low level deployment details of the architecture, which taken together, is referred to as Functions-as-a-Service or Serverless computing [18, 162]. With FaaS, programmers develop only the individual microservice functions, which they upload to the platform and specify event (microservice) triggers.

Considering these advantages, we believe that the the serverless microservice model may also be useful for the domain of IoT. However, to enable this, many new challenges must be overcome. First, the cost of microservice inter-operation may be too high for resource constrained devices. First, they were designed for resource-rich settings (i.e. clouds) and as such trade off programmer productivity for performance/energy efficiency. The use of container isolation and virtualization, IPC, address space traversal, and other features may impose too much overhead for use on severely resource constrained devices. Second, microservices often use statically and strongly typed interface description languages (IDL) to specify their inputs and outputs, from which code for various programming language implementations can be generated. However, existing IDLs are designed for resource rich cloud platform and cannot be verbatim ported to microcontrollers due to their liberal use of memory allocation, exceptions, and tightly coupled network stacks. Third, all microservice and FaaS platforms today rely on general purpose operating systems, e.g. Linux or Windows, which do not run on or are too heavy weight for resource constrained and embedded devices.

With this dissertation, we investigate and and attempt to overcome many of these challenges surrounding development and deployment of multi-scale IoT applications today. We do so by building upon and extending recent advances in cloud computing, embedded systems, programming systems, and security for use in portable, end-to-end, and multi-tier application development and deployment for IoT. Specifically, we attempt to answer the following thesis question:

Is it possible to unify the cloud and IoT abstractions efficiently through

software infrastructure optimizations on the microservices model?

To answer this question, we focus on investigating, developing, and evaluating

1. a new programming model for building multi-scale applications that enables portability and programmer productivity across the devices that comprise an IoT deployment,

2. a scalable security model that can efficiently support all tiers of IoT, and

3. the design and implementation of an operating system specialized for microservices to realize this vision.

To overcome the diversity of software stacks and provide (1), we describe a novel model for building IoT applications based on microservices called Devices-as-Services, which advocate for the modeling of hardware as typed microservices that can be consumed just like a normal service. This model abstracts away most of the platforms, and if implemented well, could allow for seamless migration of programs between microcontrollers and cloud servers.

We then describe CSpot, an implementation of the Devices-as-Services model as a FaaS-like runtime that can be hosted on Linux edge and cloud machines as well as microcontrollers running a bare metal operating system. CSpot allows users to provide low level C++ or C programs that can execute in response to writes to an append only log. We then extend this using our work on EdgePy and NanoLambda to allow running python and unmodified AWS Lamdba serverless microservices across the same tiers.

To provide (2), we suggest that a fundamental rethinking of access control and privacy of services is warranted. To which end we present CAPLets, a novel approach to security, access control and privacy that can address the challenges of IoT efficiently. CAPLets makes use of a combination of capability and token based access control to all

but eliminate any storage on microcontrollers without sacrificing security. While TLS is sub-optimal in resource restricted devices, alternative primitives based on symmetric cryptography can be repurposed to efficiently secure the said capability tokens over a network with little computational overhead. The novel combination of these security and privacy primitives can unlock fully decentralized authorization that can scale down to small microcontrollers as well as up to massive cloud servers, providing a uniform authorization model for all tiers.

CAPLets' light-weight, special purpose virtual machine for implementing access control policies whose programs can be embedded in the capability tokens' bodies allows even third party users to define completely arbitrary authorization predicates to address the rigidity challenge of capabilities. A novel key exchange algorithm enables CAPLets to provide configurable privacy efficiently on microcontrollers.

Finally, for (3), we discuss Ambience, a special purpose operating system to realize our vision to its logical extreme. Ambience is designed from the ground up to eliminate all costs of general purpose operating systems and enable domain specific optimizations to provide a truly uniform programming and deployment environment across scales. Ambience can run on ARM microcontrollers with 64 MHz clocks as well as high end x86_64 processor running at multiple GHz.

The enabling novelty of Ambience is the leveraging of various statically known information to drive optimizations not only for individual services, but for the kernel images that host said services at runtime. We note that such information has always been available to existing systems as well, but instead of being used to their full potential, they are often discarded early. Ambience also provides domain specific abstractions towards microservice applications that may not be welcome in a general purpose operating system, such as the ability of hosting multiple unrelated and non-cooperating services inside the same address space to recover the cost of premature service decoupling.

In summary, we answer the thesis question via the following research innovations. We present a vision for building scalable IoT applications across tiers based on the microservices architecture, and an initial implementation via CSpot. Next, we describe CAPLets, an efficient, capability based access control mechanism which covers all resource tiers. We discuss its truly flexible policy definitions and novel key exchange algorithm, as well as its cryptographic construction. Finally, we specify the design and implementation of Ambience, a from the ground-up operating system that provides uniformity to microcontrollers and servers alike.

The remainder of thesis is structured as follows. Chapter 2 provides an overview of the related literature on approaches to IoT programming, security and communication as well as cloud microservices and operating systems. We discuss the benefits and limitations of these technologies and examine the state-of-the-art systems. In Chapter 3, we introduce Devices as services, a structured and principled method for programming IoT applications uniformly. Also in Chapter 3, we discuss CSpot, EdgePy and NanoLambda which implement this model practically to allow the execution of C++, C and Python programs on microcontrollers and cloud servers alike. In Chapter 4, we detail CAPLets, a capability based approach to security and privacy, designed and implemented with power aware primitives in mind to allow execution on even the weakest microcontrollers. In Chapter 5, we discuss *Ambience*, a brand new operating system for the efficient execution of microservice applications across resource tiers to overcome the uniformity challenges of depending on divergent software stacks. Finally, we discuss future work and conclude in Chapter 6.

Chapter 2

Background and Related Work

2.1 Programming the Internet of Things

Internet of Things (IoT) applications today are built through the adaptation of existing, proven and scalable internet technologies to the varying resource tiers. For instance, commercial cloud providers' IoT offerings consist of a low level program running on a real time operating system (FreeRTOS [70]) running on the extreme edge, communicating via MQTT [31] over TLS [154] over TCP/IP [2] with a Linux system running on a single board computer such as a Raspberry Pi [156], which in turn communicates with serverless [20, 24, 76, 141] microservices running on powerful cloud servers, communicating over RPC protocols [79, 165, 160].

Such amalgamation of incoherent (and often incompatible) technologies poses multiple challenges and inefficiencies to the development, deployment and security of IoT applications.

Some cloud providers attempt to provide a common runtime [78, 131] for the cloud and the edge to enable code sharing and facilitate secure and reliable communication. In such a design, while the cloud and the edge machine can share code potentially in a

high level language, the microcontroller side is often written as a separate, embedded C program that has to correctly and securely interface with the rest of the system.

There have been some efforts [56] at providing a uniform programming environment through the porting of Linux to the extreme edge, however, even the most stripped out Linux can work on only high-end, expensive microcontrollers [129, 128]. Such microcontrollers are often employed for multimedia applications as opposed to cheap sensing devices, making their use in IoT applications infeasible.

2.2 Security & Access Control

As the name implies, most IoT devices communicate over a network, often a wireless one. To avoid tampering and eavesdropping, such communications must be cryptographically secured. The mainstream approach to securing this link has been to again take what works in the cloud and use it on a microcontroller. TLS [154] (Transport Layer Security) (and SSL(Secure Socket Layers)) has served the internet community well for multiple decades. However, TLS is based on memory and compute intensive asymmetric cryptography primitives such as RSA [105] and elliptic curves [61]. While their use in consumer machines is inconsequential, even on highly concurrent cloud servers, TLS termination is a significant cost and there has been significant interest in reducing TLS processor load through hardware offload [146, 74]. On microcontrollers, full TLS hardware offloading is not a reality, and it often has to be implemented all in software [147, 23, 125]. On top of causing code bloat and precious RAM waste, such implementations take upwards of 10 seconds to perform a single TLS handshake. While in absolute numbers 10 seconds could be tolerated, every second spent performing TLS operations is a second spent not sleeping, reducing battery life.

TLS based systems also require the maintenance of a certificate storage [6], which

requires periodical revocation checks and as each certificate takes multiple kilobytes to store, a sizable storage on the microcontroller itself.

Even with this costly setup, TLS can only implement *authentication* using public keys, that is, it can prove a party is party X. However, authentication by itself cannot implement sufficient authorization. Once a party authenticates themselves, what permissions they have are determined by the use of Access Control Lists (ACL), Role Based Access Control (RBAC), Attribute Based Access Control (ABAC) or similar. All these approaches require extra storage on the microcontroller themselves, incurring an extra storage bloat. Storing the tables at a cloud server introduces centralization to a system, leading to availability and reliability problems.

Capability based access control has seen some interest from the community as it can enable a truly decentralized, low-resource load alternative for authorization. Network capabilities (e.g. [135]]) make use of cryptographically protected tokens that entitle their owners the permissions the token carries. Since the access rights are carried *within* the tokens, they can be stored by the clients instead of the server, relieving it from storing potentially unbounded number of tokens.

Capability and token based access control have been considered in an IoT context by [124, 92, 65, 90]. However, all of them still depends on the use of expensive TLS primitives to secure their network.

Authors of [186, 143] advocate for the use of blockchain technologies to secure IoT devices. Due to the processing and storage requirements of blockchain technologies, such approaches are fundamentally incompatible with resource-restricted devices and none of the approaches are evaluated on devices weaker than Raspberry Pi level hardware, a comparatively powerful machine in IoT deployments.

Another approach taken by multiple authors [110, 173] have been to hide IoT devices behind more powerful gateways that can implement traditional security practices. To

make economical sense, many resource-restricted devices must be handled by a single gateway. However, this sort of deployment create a single point of failure where multiple nodes can become unavailable due to a failure on the gateway. Also, while the gateways perform access control and forward correct requests to devices, the devices still must perform some rudimentary authorization to prevent forged requests from attackers.

2.3 Communication

Following the trend of porting existing technologies to IoT, most platforms and applications make use JSON [11] as their wire format. As parsing and emitting JSON is a non-trivial task, microcontroller sides of such applications are either bloated, incorrect or insecure, if not all. Use of more efficient binary protocols [43, 149, 75, 160, 165] is gaining some traction, but not commonplace in IoT systems yet.

Wire formats have to be transported over some network protocol, and prior work [25, 98, 78] makes heavy use of message queuing as a decoupled, asynchronous communication infrastructure. MQTT [31, 169] is often the specific protocol of choice and is supported by all public cloud vendors' IoT clouds. Use of MQTT makes services topology-agnostic by placing an indirection over the MQTT broker no matter where the publisher and subscriber are deployed. However, this uniformity comes at the cost of performance, latency and battery life when two services are co-located on the same machine and could communicate over much more efficient protocols.

2.4 Cloud Microservices & Operating Systems

IoT backends running on cloud servers are often architected as a composition of small services running in isolation. Services communicate with each other over RPC channels.

Such an architecture provides maintainability, scalability and fault isolation. These systems still run on commodity, traditional operating systems, often Linux. However, the decoupling of services across address spaces and communicating over reified message channels, combined with the traditional IPC/RPC mechanisms [155, 2] provided by existing systems introduce a bottleneck. Indeed, microservice deployments are deemed far less efficient than their monolithic counterparts, but still worth the trade-off.

As the flexibility to deploy services on remote machines as well as locally is desirable in a scalable network server context, microservice communications are often implemented with TCP/IP networking. Use of expensive networks when a local IPC primitive will suffice is sub-optimal. Using a new serverless microservice framework, Jia Et al. [101] automatically promotes qualifying service dependencies to use pipes to recover some of the overheads of typical microservice platforms.

As the services, and the associated user space components, perform less and less work, the inefficiencies of the operating system starts to dominate the overall runtime. The performance of the networking stack, the scheduler, access control, IO and IPC subsystems becomes more and more important.

The research community has so far evaluated a variety of approaches to these kernel inefficiency challenges. Kernel bypass methods [34, 148, 60, 100] attempt to eliminate the kernel from the data path as much as possible. For instance, in such systems, the network cards (we discuss networking as an example, but a similar approach applies to other subsystems as well) deliver data directly to the user space memory instead of traversing the kernel, and traditional kernel responsibilities are either handled by the user space or eliminated entirely. This way, a user space application can use essentially a special purpose network stack instead of relying on the general purpose kernel one.

Alternatively, Unikernels [122, 44, 140, 113] merge the application and the kernel to again achieve the special purpose network stack the application needs. However,

Unikernels do away with hardware isolation of individual components they host. Either all services must be linked to a single address space without fault isolation, or services must be deployed in separate virtual machines, preventing the exploitation of colocation.

As microservice applications consist of a large number of independent services, starting and maintaining a healthy microservice deployment is a significant challenge. To ensure correctness at all times, microservice frameworks [112, 58] employ declarative deployment manifests, that is realized and maintained by a control plane as opposed to manual launching and monitoring of services. In such systems, the control plane constantly monitors the cluster of services and the desired state of the deployment and eventually matches the state of the cluster with the desired one. We observe the manifests contain more information than what state of the art systems can take into account. For instance, Kubernetes knows what ports a container or pod will open, but simply discards this information rather than take advantage of it to drive optimizations.

Chapter 3

Devices as Services

3.1 Introduction

As the Internet of Things (IoT) grows in size and ubiquity, it is becoming critical that we perform data-driven operations (i.e. analytics, actuation, and control) at the "edge" to reduce the latency, response time, cost, and energy use for IoT applications. As such, edge systems increasingly co-locate data management and analysis services with sensing, instead of requiring that devices ship their data over long-haul networks for remote processing using the traditional "cloud" model.

Edge systems [31, 169, 63, 161, 25, 78, 69] typically implement a publish/ subscribe (pub-sub) model in which devices publish streams of data (often via a nearby broker); when supported, actuation often uses a separate protocol. Alternatively, some solutions target a client-server model, where the IoT devices are the clients that respond with data whenever a decision making server needs it. In our view, a client-server model is well suited to emerging IoT applications in which resource constrained edge devices provide *nanoservices* – data-driven actuation and control, data analysis, processing, *and* sensing.

We believe that the IoT deployments in the not-too-distant future will include multi-function devices. As a result, a services model in which the specific function (including data publication) can be requested from each device when it is needed is more appropriate. Further, because devices will continue to be resource constrained, they will require "helper" services at the edge that augment device capabilities and enable scale. In this paper, we outline this approach to implementing *Devices-as-Services* and describe some of the capabilities of an early prototype.

Our work is motivated by the following observations.

- IoT applications can and will likely be structured as collections of services that require functionality from a device tier, an edge tier, and a cloud tier
- in-network data processing can significantly reduce response time and energy consumption [188],
- edge isolation precludes the need for dedicated communication channels between application and devices, and facilitates privacy protection for both data and devices,
- actuation in device tier will require some form of request-response protocol where the device fields the request,
- the heterogeneity of devices militates for a single programming paradigm and distributed interaction model, and
- multi-function devices can and will be able to perform their functions (including data publication) conditionally based on application needs (e.g. as a power optimization).

Thus we propose a new model for distributed IoT applications in Figure 3.1. We *flip* the client-server model such that devices at the edge are "servers" that although resource

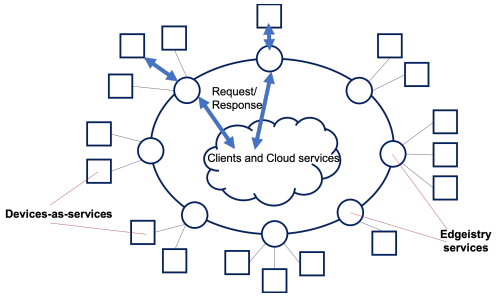

Figure 3.1: *A new distributed services model for IoT: "client" applications in the cloud compose services exported by resource-constrained devices at the edge via Edgistries – edge nodes that facilitate device discovery/registration, privacy mediation, and optimization.*

constrained, service multiple, scalable applications (i.e. clients) deployed in the cloud. Note that in many commercial solutions [78, 25] devices "publish" data to channels to which servers (running in the cloud) subscribe. We propose to move the servers to the edge (as a way to scale data ingress, lower latency, and improve privacy) thereby reversing the current cloud-centric architecture. Such a model requires a new distributed architecture that leverages edge resources in these ways.

In this paper, we define one such architecture, investigate how it can accommodate this new model, and evaluate the implications of its use in IoT settings and for servicing IoT applications. The architecture is based on the functions as-a-service (FaaS) programming and deployment model (the execution engine that underpins serverless computing for clouds [126, 88, 121, 115]), and integrates a new approach for efficient client and server discovery, authentication, and authorization via a combination of capability-based security [135, 42, 81] and a set of edge services for service registry, privacy mediation, and optimization called an *Edgistry*.

We prototype an early implementation of this architecture and approach using microcontrollers, single board computers, and edge clouds. We empirically compare the energy consumption and performance of traditional and flipped architectures, as well as our capability approach versus TLS/SSL based mechanisms for authentication. We find

that it is possible to implement Devices-as-Services – a *flipped* model of the currently popular cloud-based IoT architecture – efficiently, using FaaS as a universal programming paradigm and a set of application agnostic edge services.

3.2 Devices-as-services – a New Approach

Our view is that the current Internet and cloud architecture should "reverse" to accommodate scalable IoT applications. Rather than hosting services in the cloud (logically making IoT devices clients of those services), we view devices as "servers" and applications running in the cloud as "clients".

This viewpoint is supported by several observations. First, devices are resource restricted making them capable of delivering limited and relatively fixed functionality. This functionality is naturally described as an enumerable set of services (responses to requests) that are composed by applications. These services are necessarily "small" but even fully resourced cloud services are now being developed as microservices [106]. Devices are the "nanoservices" in an IoT setting.

Secondly, applications must compose device capabilities from vast collections of devices into meaningful functionality for users, and in an IoT setting, the user scale will be substantial. The cloud is where this scaling will necessarily take place, both in terms of aggregating device functionality, and matching these aggregations to user demand for them.

Thirdly, much of the current technological development for IoT [78, 25] is focused on bringing devices to the cloud. Many of the commercial offerings provide an SDK for using MQTT [31, 169] or AMQP [3] to publish telemetry, events, etc. to objects (which subscribe to device channels) in the cloud that represents each device. For actuation, each device must separately subscribe to such a channel to receive asynchronous commands

(although for low-power applications that use MQTT-SN, this option is not possible). With our system, the same server code runs at all tiers — device, edge, and cloud — unifying the device API as a first-class services API.

The currently prevalent commercial IoT/cloud APIs require devices to act separately as "publishers" or "subscribers" or both (the last to implement "closed-loop" actuation). We believe that IoT infrastructure must accommodate a richer model in which devices (however resource restricted) are capable of actions as well as event telemetry. Thus, logically, IoT devices are better modeled as servers that provide services to applications.

3.2.1 The Edgistry

Our approach splits the provisioning of services between devices and a common set of management services located at the edge, termed *The Edgistry*. Devices host small, resource-light services that field requests from, and respond to, client applications (hosted in the cloud or edge). The Edgistry

- implements an eventually consistent distributed service registry (e.g. using blockchain consensus protocols such as Tendermint BFT [46] and Hyperledger Fabric [8]),
- acts as a speed-matching communication service,
- can protect device privacy (e.g. through anonymization) by isolating the service provider from service consumer, and
- provides computational and storage off-loading services (e.g. content caching) for device-hosted services.

From a programmability perspective, we advocate a universally portable "Functions as a Service" (FaaS) capability [103, 126, 88, 121, 115]. Services on the device are implemented using a "micro FaaS" – a small, specially tuned, implementation of FaaS

specifically targeting resource-restricted microcontrollers as an execution platform. The micro FaaS supports the same programming APIs as a heavier-weight (and serverless) implementation designed to run on an edge device, in a private cloud, or in a public cloud. In this way, IoT applications can be coded using a single "FaaS everywhere" set of programming abstractions from device to cloud.

To enable this portability and also data durability in a distributed setting, such scale-spanning FaaS capability must define a portable storage abstraction, ideally having append-only semantics. Thus all computations, regardless of location, can persist data by appending it to some object hosted in the device-edge-cloud hierarchy using a common API.

3.2.2 Request Forwarding and Duty Cycle

Note that to save power, some devices will need to spend most of their duty cycle in a power-saving "sleep" mode. For example, an ESP8266 microcontroller [66] 0.01 mAh (milliamp-hours) in deep sleep mode and requires 320 mAh to transmit a packet using its on-board WiFi. Using a 18650 Lithium-ion rechargable battery, it is possible to operate the microcontroller for approximately 1 year without a battery recharge if it limits its communication to every 10 minutes (on the average, using WiFi and WPA2) [178]. Thus, the network connectivity we can expect will be initiated by the device on a duty cycle dictated by its power budget.

Note that in a "typical" server setting, the client initiates a connection to the server when making a request. Thus the server's activation can be triggered by the connection initiation. In this setting, the connection and the request-response interaction are decoupled. That is, the device (running a service) polls the Edgistry to see if in-bound requests are queued there. Any pending requests will be forwarded to the device which

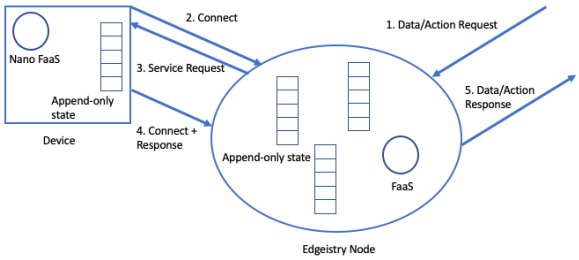

Figure 3.2: *Speed-matching service requests via an Edgistry*

can then disconnect (to save power). When a response is ready, the device connects to the Edgistry and sends a response. Moreover, both the service running on the device and the proxy running on the Edgistry uses append-only data structures to store state waiting for successful transmission.

Figure 3.2 shows how a request for service from a device (registered in a serachable registry, not shown) arrives at the Edgistry and is appended to a request queue associated with the device. When the device contacts the Edgistry (as part of its duty cycle), the request is forwarded to the device and appended to a queue of requests, thereby triggering a function in the micro FaaS. When the response is ready, the device connects and sends it to the Edgistry where it is appended to a queue of responses destined for a client. Finally, the Edgistry forwards the response to the original requester.

3.2.3 Device Registry and Privacy

The Edgistry must implement service discovery (i.e. for devices implementing services) and client registry. That is, when a client makes a (possibly speed-matched) request from a service on a device, the Edgistry will need to keep track of where the response must be returned. We envision each device pairing with a one local Edgistry node (or a small number of local nodes) so that it does not have to store per-request client tracking

information.

Service discovery can be implemented using a distributed, eventually consistent blockchain consensus protocols such as Tendermint BFTT [46]. A blockchain consensus model is particularly attractive because it offers the opportunity for the Edgistry to implement privacy features such as location and identity anonymization [120]. For example, an application may wish to contact a device within some geographic region without the need to know the precise location of the device. This type of location "fuzzing" can be implemented using policy delegation by the device to the Edgistry node with which it is paired. By hiding the identity of the device from the service requesters, we can mitigate denial of service attacks as the Edgistry delegates the service to one of the devices in the network in a geographical neighborhood without revealing the device identity to service consumers.

3.3 Capability-based Service Access

Security is a challenge for this inverted model because it requires distributed policy implementation governing a vast number of services, each implemented using the barest minimum of computational and storage resources. This latter requirement alone precludes the use extant public-key/private-key "standards" for securing client-server interactions. Specifically, asymmetric cryptography computations and key storage are costly for resource restricted devices. Moreover, current TLS [154] protocols do not allow servers to control packet size, so servers must have large memories (e.g. 16k for half duplex, 32k for full duplex) to conform.

Our approach uses a distributed, capability based authentication scheme to address these challenges. A capability is a communicable, unforgeable token of authentication. It encapsulates an object, the access rights on that object, and a cryptographic signature

that maintains the integrity of the capability. We implement access rights as a bitmap and generate a signature using a private key based method such as Hashed Message Authentication Codes (HMACs) [111]. The signature protects the object name and access rights in the capability. To verify the token, the server receiving a request regenerates the signature from the capability body and compares it against the capability signature. If it matches, the server executes the request and discards the capability.

3.3.1 Controlled sharing without the server

Capabilities can support privilege reductions without involving the server. For example, Amoeba [135] implements a derivation mechanism that uses commutative hash functions on access-right bitmaps to selectively reduce capability rights.

We generalize this derivation mechanism to support a wider variety of policy implementations. To enable this generalization, each capability maintains a derivation history. Using this history, the server can verify the validity of derivations. On each derivation, the signature of the current chain is merged with the hash of the new capability. Upon receiving a request, the server walks the chain applying the hashes. If the signatures match at the end and each derivation is legal (e.g. does not add new rights), the derivation is deemed valid, and the request is served. This mechanism is similar to Macaroons [42] for web services, but with optimizations that permit multiple entries to be combined in a single signature generation. Because any holder of a capability can extend the chain of rights constraints this capability system can implement truly distributed access control.

It is possible for our derivation chains to grow over time. To bound this growth we provide a flattening functionality, which compresses a chain of derivations into a single capability upon each successful request. Because server verification of capabilities is efficient (cf. Section 4.6) this methodology is appropriate for very large scale IoT

deployment.

3.3.2 Protecting Capabilities

To avoid the use of resource-consuming encrypted links between devices at the edge, we devise a novel approach to protecting capabilities in clear text channels. In our scheme, capabilities are further constrained to a specific request before being transmitted. Thus, a capability transmitted over the network can only be used for that request. We number requests with monotonously increasing sequence numbers to prevent replay attacks. To enable this, we employ the current time as the sequence numbers of each request. Since updating the sequence number in the server requires a valid capability, a DOS attack cannot be employed by an outside attacker. The device stores the last sequence number for efficiency purposes. If a client has a buggy time protocol implementation or a clock drifts, our protocol handles this case by returning the current sequence number in the error response. The clients can synchronize their clocks using this number. Upon the detection of such a drift, an external service such as the Edgistry can fix the sequence number using a special capability. If the attack model of the application involves malicious clients, a separate sequence number can be employed per client.

3.3.3 Maintaining trust

The end device and Edgistry perform 2-way authentication so that either can verify requests from the other. During bootstrap, the end device passes a capability to the Edgistry. Using 2-way authentication, the Edgistry also passes a capability it signed to the end device. Using this capability, the Edgistry can verify the authenticity by requiring the end device to send responses by deriving from that capability.

3.4 A Prototype

We prototype this "flipped" client-server model and our capability-based security method using CSPOT [181], an open source [1] distributed platform for deployment and execution of IoT applications. CSPOT runs as a FaaS server on microcontrollers (i.e. as a micro FaaS), and as a serverless FaaS runtime system on edge devices, private clouds, and public clouds. In CSPOT, a function invocation *can only* be coupled with a "Put" of data targeting some storage object, which is append-only and persistent. The function coupled with the put has direct access to this newly posted data as well as all previously appended data up to some limit specified with the object is created. CSPOT data objects are called WooFs (**W**ide-area **o**bjects **o**f **F**unctions) and each WooF resides in a single namespace. Namespaces are addressed by a URI and, thus, network accessible. A CSPOT application can only persist data in WooFs located in one or more namespaces – the functions that are invoked are thus stateless while executing.

In our prototype implementation, we use capabilities to authenticate four CSPOT functions: namespace creation, WooF creation within a namespace, Put operations which store data in a WooF (and may invoke a function), and Get operations which retrieve data from a WooF.

Upon flashing a device, we generate an HMAC secret. A "root" capability is securely issued when the device is paired with an Edgistry node. This capability carries the right to create a namespace on the device which the Edgistry node uses to initialize the service.

As part of the bootstrapping process, the Edgistry creates the necessary namespaces and the WooFs on the device. After the CSPOT components are initialized, the application, for instance a temperature monitor, is started and periodically Puts a new reading into the local WooF.

[1] https://github.com/MAYHEM-Lab/cspot.git

Note that because the Edgistry has the initial root capability, it can implement all access control policies via derivations from the root capability once the initial trust relationship is established. For instance, the Edgistry can generate capabilities for the WooFs and namespaces it creates during the bootstrap process, rather than the device generating and transmitting them. The device need only verify each derivation, thereby saving code complexity and power.

At the end of the bootstrapping process, the Edgistry derives the application capabilities. For instance, the Edgistry has full access rights to the WooFs in the device, but it derives and transfers a read only capability for the clients (applications) to use, following the principle of least privilege.

Policy delegations also can be performed by clients. For instance, an application client A can share a capability C_1 with another application B with a constraint that the derived capability C_1' is only valid in the presence of another capability, C_2, that B must present alongside C_1'. Thus clients can also create derivations that implement policies such as identities (represented by an identity capability) without the server's involvement (i.e. without contacting the device which can always verify *any* derivation).

3.5 Evaluation

Our initial goal is to verify the execution and power efficiencies of this new model at the device level. We begin with an abbreviated comparison of the Devices-as-services model to the IoT infrastructure offered by Amazon AWS [97]. Table 3.1 shows the end-to-end timings from a ESP8266 [66] microcontroller to AWS using our new model, and the AWS IoT SDK coupled with AWS Lambda.

This benchmark uses NTP-synchronized clocks to record a timestamp on the device and another in AWS; it computes end-to-end latency as the difference between the two.

Table 3.1: Comparison of Devices-as-services and AWS IoT. Units are milliseconds; across 100 benchmark runs.

	mean	stdev	max
AWS IoT+λ	5578	265	6843
CSPOT device-¿Edgistry-¿AWS	608	5.78	652

Both CSPOT and AWS Lambda implement an event-driven FaaS programming environment. For AWS, we persist in DynamoDB [54] For the CSPOT case, we replicate data on an Edgistry node (an Intel NUC) [96] in a WooF before forwarding/persisting it in CSPOT running in a virtual machine in AWS. The table shows mean, standard deviation, and maximum latencies in milliseconds over 100 experimental runs. The Edgistry node is hosted using an edge cloud [63] running Eucalyptus [139] located in the same room as the microcontroller. The edge cloud is connected to the UCSB campus network. From this data (and a more detailed comparison that includes Azure IoT Hub [181])it is clear that this new methodology is at least an order of magnitude faster (in terms of latency) than comparable commercial offerings even when it replicates data in the Edgistry. Further, the coefficient of variation for the AWS experiment is 0.04 and 0.009 for the CSPOT experiment, indicating that our system is also an order of magnitude more stable in terms of performance variation.

Moreover, devices that publish all of their data all of the time (e.g. using MQTT to leverage AWS or Azure) wastes precious battery lifetime when that data is not demanded. This new model allows devices to function as servers and only to respond when data is requested by a client application. Thus, in this work, we focus on improving the efficiencies of the security features at the edge, where computational power and electrical energy (i.e. battery power) are at a premium.

Much of the latency experienced in commercial offerings (and the concomitant loss of

Table 3.2: Comparison of cryptographic algorithms for signing and verifying a 32 byte message on the ESP8266. Average execution time and standard deviation (in parens) are shown. Last 2 rows show end-to-end measurements.

Algorithm	Sign ms (stdev)	Verify ms (stdev)
PKCS1 (2048 bit)	3280 (190)	187 (4)
PKCS1 (4096 bit)	31580 (190)	9190 (9)
ECDSA (256 bit)	214 (1)	4340 (216)
HMAC (64 bit)	0.37 (0)	0.37 (0)
HMAC (128 bit)	0.37 (0)	0.37 (0)
3-level Derived Capability (64 bit)	0.77 (0)	1 (0)
5-level Derived Capability (64 bit)	1.18 (0.04)	1.3 (0)

battery life through additional active time in the device) is associated with the TLS-based security protocols that MQTT and AMQP implementations use. While it is possible to use the bidirectional nature of MQTT and AMQP communications to implement close-loop device interactions, these protocols (let alone the APIs that actuate them) are not specifically designed to implement higher-level service interactions on devices with moderate or severe power restrictions and/or very limited memory. Devices-as-services requires a more efficient, high-level interaction and key to that interaction is efficient authentication. We next compare our capability-based approach (which uses an HMAC-based mechanism) to competitive approaches. In all experiments, SHA256 hash was used for generating a digest from messages. For the public key cryptography experiments, we use the recommended key size for RSA of 2048 bits and 4096 bits and 256 bits for ECC. For the hash generation, we use a custom version of libemsha. Generation of SHA256 hashes from messages are common to both approaches. Our implementation takes around 88 (0.06) microseconds per SHA256 hash on a message of 32 bytes. The time it takes to hash a message scales linearly with message size.

Table 3.2 shows a comparison of the execution times for various cryptographic tech-

niques when used to sign and verify messages. The table shows the times for RSA (using PKCS1) for two different key lengths, ECDSA (an Elliptic Curve Cryptography – ECC – method popular in many IoT applications [90]), and an HMAC-based scheme we have implemented. Below the double lines we also show the performance of a 3-level capability derivation and a 5-level derivation on the server (e.g. to represent a setting in which the clients constrain capabilities multiple times). We also ran a TLS based server experiment. We find that each connection uses 3950 milliseconds (ms; standard deviation (stdev) 11 ms) and 32320 ms (stdev 5 ms) for a simple TCP request and 2048 bit and 4096 bit keys, respectively. The difference between these two measurements is due to network performance variability.

Clearly, an HMAC-based approach is considerably less computationally intensive than either of the competitive approaches. Indeed, even the derived capability timings are better for a 3-level derivation than for a single capability verification using either of the other schemes.

While the results in Table 3.1 indicate that, overall (even with additional persistent data replication) our method requires an order of magnitude less "active time," and active time is proportional to power usage, we highlight on the energy efficiency of the authentication protocol. Specifically, our method uses 0.072 mJ and 0.185 mJ for generation and verification, respectively. RSA signatures (2048 bit keys) consume 606.1 mJ and 34.6 mJ, and ECC signatures use 39.54 mJ and 80.32 mJ, respectively. That is for capability generation our method uses between 3 and 4 orders of magnitude less energy for authentication than either RSA or ECC.

3.6 Summary

We propose a new "flipped" client-server model for IoT in which devices at the edge are servers that provide nanoservices, which applications in the cloud (the clients) compose for their implementations. We contribute a novel approach to distributed service design based on "FaaS everywhere," edge-level support, and a novel capability mechanism for distributed policy implementation, a fuller exposition of which is available from [182]. Our empirical evaluation shows that this approach is feasible and introduces very little overhead and power consumption.

Devices-as-services provides a principled and structured way of programming heterogeneous, distributed and scalable Internet of Things applications to conclude our first research direction laid out in Chapter 1, a significant advance over the existing mainstream practice of amalgamating incompatible technologies.

Chapter 4

CAPLets

4.1 Introduction

While Devices-as-services provide a uniform method of programming complex IoT applications, the implemented services must be available over a network (after all, it is the *Internet* of Things). Devices-as-services on its own does not define any security mechanisms, and it is left to the particular application. However, security in the context of IoT turns out to be very difficult.

Existing IoT programming systems either focus on a specific IoT resource tier (i.e. the cloud), or attempt to repurpose and amalgamate existing tools and protocols not designed for the resource constraints, intermittent connectivity, and failure frequency of IoT deployments [21, 131, 78, 132, 35]. For example, most end-to-end IoT systems [31, 169, 63, 161, 25, 78, 69] use a publish/subscribe (pub-sub) model in which devices publish streams of data (often via a nearby broker); when supported, actuation often uses a separate protocol [138].

Similarly, for access control, many IoT systems use Transport Layer Security (TLS) [154] protocols, based on public-key cryptography [97, 25, 98]. These systems use edge devices

and microcontrollers to communicate with servers and edge proxies via encrypted channels, often using RSA certificates. Although TLS addresses many security challenges, it also consumes significant resources on resource restricted devices (memory, computation, network, battery power, etc.) and depends on a number of resource-intensive operations for its security. The latter includes accurate and secure time keeping, awareness of certificate revocations, and maintenance of root certificates, among others.

Due to the implementation complexity, resource consumption, and configuration difficulty of TLS-based security, many IoT devices have weak or no access control, and are easily compromised [9, 119]. Moreover, the heterogeneity of device-maintenance interfaces and the complexities of remote device management often prevent users from updating weak or faulty implementations when improved software is available. As a result, many devices are placed behind gateways or firewalls which proxy requests when they are actually deployed [110, 173], rendering the resource expense associated with TLS-based security on the device needlessly redundant and costly.

Our goal, with this work, is to address these challenges at the device level. A full CAPLets IoT deployment uses TLS to protect client-to-client communications where clients are hosted on relatively resource-rich platforms, but removes the need for resource-restricted devices to implement or proxy a TLS connection.

The primary problem with current cloud-based approaches is that they make resource-restricted devices act as network-attached clients that access services hosted on resource-rich platforms. Because these services use technologies that support Internet web-services (e.g. in the cloud) they define security protocols that do not account for the paucity of on-device resources at the edge. As a result, devices acting as clients must use (or proxy the use of) the resource-intensive protocols mandated by the services. That is, the services define the protocols that the clients must use and, because they are often repurposed cloud-based web services, these protocols impose heavy resource loads on client devices.

Due to this model, current applications are dependent on the availability of cloud services at all times for users to access their own devices. For instance, a smart-light needs to be connected to the cloud service even when it is in close proximity to the user. This causes temporary or permananent outages when Internet connectivity is lost [36, 99] or the manufacturer no longer supports the online service.

In this paper, we investigate the potential of inverting this relationship so that devices become *first-class* – that is, devices host services to which resource-rich clients (on cloud servers or smartphones) make requests. Such an approach requires the system software and its protocols to support client and server hosting on *any* device in an IoT deployment (from sensor to cloud). Unlike prior work, which focuses on system portability for IoT [164, 67, 183], we propose a unifying set of security protocols, called CAPLets, that complement this previous work to realize first-class devices.

To enable efficient access control in all IoT tiers including low-end microcontrollers, CAPLets replaces asymmetric digital signatures with fast symmetric MAC tags based on Macaroons [42], introduces a cheap key exchange mechanism, eliminating [1] TLS for client-server communications, and significantly reduces the storage footprint associated with authorization and certificate management. As a result, services can be hosted on sensors, on edge systems, or in a cloud. Unique to our approach is a constraint mechanism that is programmable and sufficiently flexible to represent diverse policies. As a summary of our contributions, CAPLets

- defines a capability mechanism that is sufficiently efficient for use by the least capable IoT devices (sensors, microcontrollers, battery-powered single board computers, etc.) as well as by more capable edge and cloud systems;

- defines a derivation process that augments existing mechanisms with semantic ca-

[1] CAPLets depends on TLS for client-client communications (e.g. to transmit an identity token). However, it can replace TLS for client-server interactions in an IoT application.

pability derivations to enable superior flexibility of controlled sharing;

- expands what a capability token can contain through the use of static and dynamic *constraints* to permit a wider range of policies to be expressed and to provide a protected metadata channel;
- uses metadata channels to offload policy implementation to trusted delegates across the deployment, e.g. to authenticate users and issue capabilities that a device can independently verify;
- defines an efficient and secure service request construction mechanism that does not depend on encrypted channels;
- defines a secure key exchange protocol by exploiting the cryptographic token construction; and
- supports efficient request validation, bootstrapping, capability sharing, and revocation.

We demonstrate the claimed efficiencies with an empirical evaluation of CAPLets that we conduct using microbenchmarks and an implementation of CAPLets for an end-to-end, first-class devices deployment, using a portable IoT operating system. Our work shows that CAPLets is an order of magnitude faster and more energy efficient compared to existing state-of-the-art IoT authorization systems.

4.2 Background and related work

Security is a challenge for end-to-end IoT systems because it requires distributed policy implementation governing a vast diversity of devices (in a multitude of trust domains) that may have limited computing, storage, network, and/or power capacities. In

particular, TLS and asymmetric cryptography computations and key storage that are necessary to access most web services are costly (in terms of execution time and energy expenditure) for resource restricted devices. In addition, these public key cryptography methods can only be used for authentication in conjunction with some other mechanism for authorizing the authenticated users (e.g. Access Control Lists or Role Based Access Control). All such mechanisms require some storage capacity on the device.

Capability systems are popular for controlling access to distributed resources (e.g. web services [42], operating system processes [135], and IoT [81]) because they are inherently decentralized. Such systems use unforgeable tokens of authority to grant client access to resources and to facilitate sharing of access between clients. They can also implement the *Principle of Least Privilege* [159] (entities possess only the access rights that they actually require), which is appealing in any security context.

Capabilities describe access rights associated with a resource in a way that is forgery and tamper-proof, often using a cryptographic signature. A concrete example of a capability is a path and permission bits in a filesystem: $Capability = (\texttt{/home/alice}, \{\texttt{R,W,X}\})$. A signed capability, called a *token*, can be securely shared across a network.

To verify the token, the server for a protected resource, upon receiving a request carrying the token (which the server generated previously and signed with a secret key known only to the server), regenerates the signature from the capability body and compares it against the token signature. If they match, the server "believes" the access rights carried in the capability and executes the request on the resource if the rights permit the access.

Capabilities can support privilege reductions, termed *derivations*, without intervention by, or cooperation with, the server. For example, Amoeba [135] implements a derivation mechanism that uses commutative hash functions on access-right bitmaps to selectively reduce capability rights.

Much research has examined the use of capabilities as the basis of IoT security [82, 123, 91, 80, 187, 186] however relatively few investigations have focused on the efficiencies necessary for microcontroller deployment. The authors of [91] explore the use of ECC on a 32-bit microcontroller. Our findings in this paper show that CAPLets is two to three orders of magnitude more efficient.

4.3 Threat Model and Assumptions

CAPLets makes specific (and somewhat common) assumptions about the network and physical security of the devices. We list them here to set a common frame of reference for the rest of the paper.

- Secrets held on devices cannot be remotely compromised.
- Recovering the Message Authentication Code (MAC) secret from a tag and a plaintext is infeasible.
- The randomness used for generating cryptographic secrets is not guessable by external attackers.
- An attacker may control every part of the network, including delaying, repeating, dropping, inspecting and modifying packets at will.

4.4 Mechanism

In this section, we describe core CAPLets abstractions. A **capability**, denoted $C_{Type}(data)$ or just C when their content is irrelevant, is a typed object expressing a privilege. For instance, $C_{Dir}(read, /var/log)$ specifies "read access to /var/log" for the directory type. A **frame**, denoted F, is a set of capabilities, which carry rights to multiple

objects as a single unit. A **token**, denoted T, is the unit of communication. It has a body, denoted $body(T)$, and a tag over that body. The tag is computed with a Message Authentication Code (MAC) function.

CAPLets generates tags using a Hashed Message Authentication Code (HMAC) [111], in particular HMAC-SHA256. We use HMACs because of their efficiency characteristics (we compare their performance against alternative approaches in Section 4.6). With this construction it is safe to transmit tokens back and forth over a network as any modification to the body of a token is detectable, i.e. tokens are unforgeable.

For each device, called an origin, we define a special token R called the root token. R only contains a special frame F_R, called the root frame and F_R in turn only has a special capability, C_R called the root capability. C_R grants absolute privilege to the device it is associated with.

The tag of the root token, $tag(R)$, is computed as $MAC(Secret, F_R)$. The $Secret$ is a random secret generated on the device during bootstrapping (described below). Currently, we use hardware with true random number generators to generate this secret. Tag generation for other tokens is also described below.

Tokens do not carry information regarding their origin device. This means the holder of a token must know (or discover through some external mechanism) the server that originally generated the token, and can process the token to grant access to protected resources to the bearer carried therein. While traditional capability implementations include the responsibility of absolutely naming objects [135], CAPLets refrains from doing so. The reasons are two fold: (i) including the *name* (a DNS name, IP address or a more complicated naming scheme) of the server bloats the token sizes and (ii) in a distributed network with several, often incompatible subnetworks, naming exact servers is not a solved problem. For instance, a global address for a server behind a NAT or in a non-IP network such as Zigbee or Bluetooth does not exist.

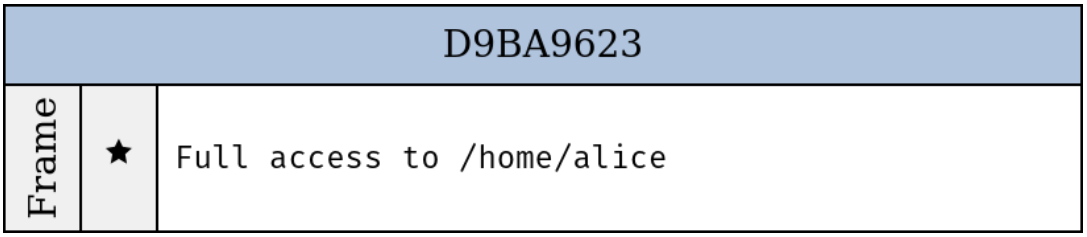

Figure 4.1: The root token for `/home/alice` directory. The bar shows the token tag, which is computed as $MAC(secret, FrameBody)$. The $\star$ denotes the capabilities in a frame.

Figure 4.1 shows an example of a CAPLets *root token*. The tag (top bar) is generated from the frame body, using a secret known only to the resource owner (e.g. a secret stored on a device).

4.4.1 Authenticated modification

A token protects its body with the invariant that $MAC(Secret, body(T)) = tag(T)$, that is, the tag carried within the token is equal to the computed tag. While tokens with this exact form are used in practice (for instance as JSON Web Tokens [1]), they preclude the ability to perform offline policy delegations.

Past work on Macaroons [42] showed, however, that it is possible to allow some controlled modification with this cryptographic construction in the form of appending to a token's body. Basically, a token starts empty with a nonce tag always available to the origin. Anyone can append new, delimited data to the token and update its tag with another MAC. Upon receiving an appended token, an origin can confirm the integrity of the whole by replaying the modifications, starting from the empty token. Upon each append, the new tag is computed as $tag(tail :: head) = MAC(tag(tail), head)$. :: is the list append operator. Note that this operation does not depend on the secret held by the server and any party can append data to a token without any involvement from the server.

In other words, every element in the token protects the next one's integrity. This

mechanism is similar to certificate verification in TLS with which a party can verify a certificate's integrity by following the signature chain starting from a Certificate Authority (CA). The difference here is that because only the server holds the origin secret, it is the only entity that can verify an entire chain.

Note that this construction is oblivious to the contents of the body. Macaroons [42] uses it to append caveats to tokens. CAPLets uses this approach but extends it to define and enforce semantic requirements associated with each append.

4.4.2 Token derivation

Specifically, CAPLets constructs token bodies as a list of frames, starting with F_R. Only the last frame of a token is used to describe its privileges. In other words, the rights granted by a token are the capabilities in its last frame. The last frame is called the *leaf frame*. The other frames are present in the token for the purposes of integrity verification and derivation checking and comprise the *derivation chain*. Formally, $body(T) = [F_R, ..., F_{leaf}]$ where $[F_R, ...]$ is the derivation chain.

We define a valid derivation to be one that reduces privileges monotonically. For instance, C_1 = "Read/write $Sensor_1$" is more privileged than C_2 = "Read $Sensor_1$". A derivation from a frame with C_2 to a frame with C_1 is deemed invalid, whereas the other way is valid. Note that this decision is application dependent and an application may define the C_1 to C_2 to be invalid as well.

A token with invalid derivations can pass the integrity check described above. Therefore, after integrity verification, CAPLets also checks that each derivation is valid in a token.

The type of a capability is used for 2 purposes: serialization and validation checking. The first is rather trivial: the type of a capability is transferred as a header in messages and during deserialization, the header is used to reconstruct the correct object. The

second is the basis of derivation validation.

Whether a derivation is valid or not is a function of two frames: $ValidDerivation(F_{old}, F_{new})$. A token is valid if all adjacent pairs in its body pass the validity check.

$$\begin{aligned} &ValidDerivation(F_{old}, F_{new}) = \\ &\forall C_{new} \in F_{new}\ \exists C_{old} \in F_{old}\ C_{new} \subseteq C_{old} \end{aligned} \tag{4.1}$$

In this equation, the $\subseteq$ operator decides whether the new capability is a subset of the old one. Therefore, the validity check ensures that each subsequent frame in a token is a subset of the previous one.

Whether a capability is a subset of another is an application defined relation between the types of the given capabilities with the special case $\forall CC \subseteq C_R$. If the relation between two capability types is not defined, it defaults to false. The subset is a binary relation between capabilities that forms a partial order. An example of the subset relationship occurs between a directory capability $C_D(DirPath)$ and a path capability $C_P(Path)$:

$$\begin{aligned} &C_P(Path) \subseteq C_D(DirPath) = \\ &StartsWith(Path, DirPath) \end{aligned} \tag{4.2}$$

This particular relationship allows the users to legally derive access rights to files in directories to which they have access. For instance, if Alice holds $C_D(/home/alice)$, she can legally derive $C_P(/home/alice/hello.txt)$.

This construction allows CAPLets applications to express flexible, natural derivations. It must be noted that the party doing the derivation checking has the authority to determine what is valid or invalid. However, sharing these relationships among the server and the clients is still beneficial to minimize non-malicious, invalid derivations.

The derivations of a token is the set of all possible tokens that can be transitively

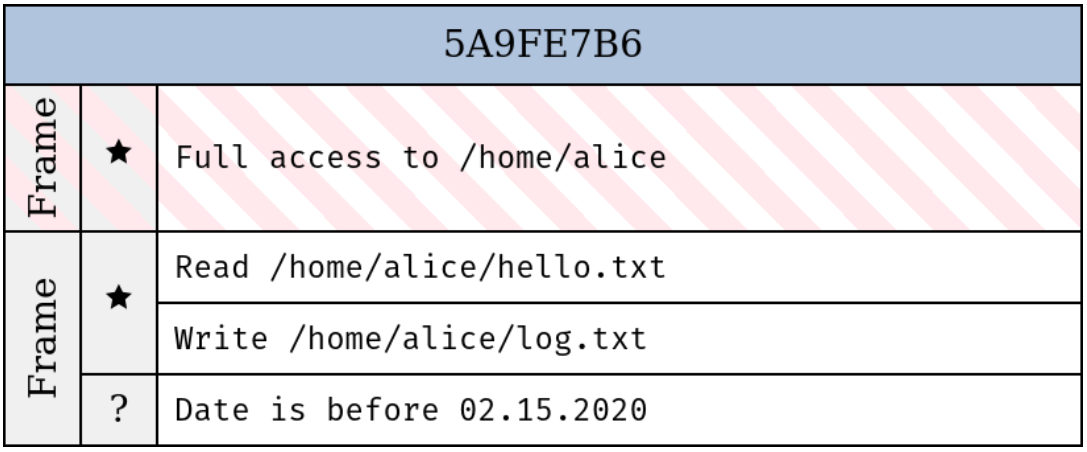

Figure 4.2: A derived CAPLets token. The blue bar displays the frame tag. The root frame, now in red stripes, is still present in the token, but its contents are only used for validation purposes. ⋆'d lines denote capabilities and lines with ? denote constraints.

derived from it and is denoted as $\mathcal{S}_T$ (for successors of T). Conversely, the tokens that can transitively derive T is denoted $\mathcal{P}_T$ (for predecessors of T). From our previous definitions it follows that $\forall TR \in \mathcal{P}_T$.

Due to the cryptographic construction, only the party who holds $T \in \mathcal{P}_D$ can compute $tag(D)$ given only $body(D)$ by taking $tag(T)$ and replaying the derivations from T's leaf to D's leaf. This property is core to how a server verifies the tag of a derived token. Upon receiving a token T, the origin replays all frames in it over R and checks whether the tag supplied in T matches the computed tag. If not, it rejects the token. After the tag verification, the server checks derivation validity (and rejects the token if invalid).

By maintaining the chain of derivations, tokens also carry within them a form of sharing history. That is, by looking at a token, a resource owner can trace how the holder of this token acquired it. As we explain below, combined with the use of identity tokens, this history allows for a powerful yet simple auditing, debugging, and revocation mechanism.

Figure 4.2 shows a derived token. The root token is at the top (the stripes showing that while it's contents are visible, they are not usable). The derived frame in the figure has two new capabilities and lacks the original one, demonstrating the subset relationship.

CAPLets' distinction between capabilities and constraints allows for efficient caching and analysis of permissions a token carries. For instance, a client can submit a token to be used in a session, which the server could verify ahead of time as much as it can (for instance time dependent constraints cannot be verified ahead of time, but endpoint constraints can). After that point, the client can efficiently perform multiple requests with that token.

CAPLets uses a zero-copy network format similar to [75, 160], improving overall verification performance.

4.4.3 Bootstrapping

To bootstrap authorization, the device generates a cryptographically random secret, which it stores in internal non-volatile memory. The internal secret is never shared externally and short of physical attacks, is unrecoverable. Using this secret with a MAC, CAPLets produces the root token R. R is then transmitted securely to the owner through the commissioning transport, typically a wired connection. For wireless commissioning, a one time secure channel is needed to transport the token. Post commissioning, the device need not create or hand out tokens (although it can).

4.4.4 Sharing

R is securely held by the resource owner, Alice. Alice uses this root token to derive privilege-reduced request capabilities for herself to use in accessing resources.

Alice can also share derivations of R with another actor, Bob. To do so, she constructs a frame with only the resources she intends to share with Bob. CAPLets appends this frame to R and computes the new tag as explained previously. She can safely pass this derived token to Bob as the MAC is one-way, i.e. even though he can see the contents R,

Bob cannot recover $tag(R)$. The transfer of the token from Alice to Bob needs to occur over a secure channel (e.g. using TLS) to prevent the token from being seen by anyone other than Bob.

4.4.5 Constraints

The mechanism explained so far is limited in that an owner of a token can only restrict it by removing a permission from it. While sufficient for some purposes, sophisticated authorization policies require the ability to condition authorization on a set of circumstances that may change after a resource is instantiated.

For example, when sharing a capability for a sensor, the owner, Alice, may want to restrict the receiver, Bob, such that he can only take readings when the device has sufficient battery power. Such a restriction is difficult to express in terms of monotonically decreasing access rights. While a special case field of required battery level can be added to every token, such an approach does not scale to support a large number of restrictions.

As a generic solution for expressing arbitrary limitations on capabilities, we introduce constraints. Constraints are carried in frames alongside the capabilities and used to decide whether or not the frame is valid. That is, if a constraint is not satisfied, the frame is deemed invalid.

Constraints consist of executable code for a sandboxed virtual machine (VM). CAPLets does not specify a VM: it could be a fully generic VM like the Java Virtual Machine (JVM), or a special purpose machine like eBPF or CapVM as described in Section 4.6. After validating a token, a server then evaluates the constraints in the leaf frame and if any constraint fails, it rejects the token.

During a derivation, an application can emit arbitrary code for the VM and add it to the leaf frame. Code contents are protected via token construction just like capabilities.

With such a construction, CAPLets deployments can be future-proof and enable the expression of truly flexible authorization policies. For instance, with Macaroons [42], the interpretation of caveats is baked in at deployment time and they are fixed and unchangeable. As a result, new policies that are needed after deployment cannot be implemented. Concretely, for the above example, if the Macaroons application did not include a *"battery level is above N%"* caveat checker at build/deploy time, it is impossible to specify such a requirement. With CAPLets, a client application can specify it dynamically.

Constraints may depend on a variety of information sources. During evaluation, a constraint has access to the contents of the tokens provided by a client and key context such as network endpoint information. They may also access global information such as the current time or battery level through the use of VM-calls. The set of global state exposed to a constraint depends on the application. For example, it is possible to authorize the constraints of a token with other tokens for enhanced protection.

Constraints naturally integrate into the CAPLets derivation mechanism. Since ordering between arbitrary code is ill-defined, the subset relation between constraints is defined to be equality. This means that each constraint in a previous frame must appear verbatim in the next one. Since all constraints must be met for a frame to be valid, adding a new one can never escalate privileges.

Constraints may also express dependencies that span multiple tokens. A token can be constrained in a way that it is valid only if the request also presents a token with a specific signature. Or it may require that a token contains a specific capability to be valid. For example, an *identity constraint* can be used to limit a token to be only valid if it is being used by a specific user. We discuss identity implementation using CAPLets in Section 4.4.8.

For widely used and common constraints, CAPLets supports a static optimization where the code for a constraint can be baked into an application in an optimized form and

can be delegated to by regular constraints. We call such constraints *static constraints*. For such cases, the frame need not carry any code for the constraint and just name the static constraint it depends on. Static constraints only carry the data they need, and are very efficient both in terms of network transfers and execution time, while maintaining the flexibility of CAPLets.

For example, a token can be constrained to be valid only when it is raining, as determined by reading an on-board rain sensor. Another example is an end point constraint which checks whether the request originates from a certain end point. For a static rain constraint, the body of the constraint is empty: its existence is enough to make a decision. For the end point constraint, its body must carry the specific end point to check at validation time.

4.4.6 Capability Revocation

CAPLets has two mechanisms for revocation with different guarantees: eventual and immediate. Eventual revocations are space efficient and CAPLets provides a timeout constraint for its implementation. Tokens with a timeout constraint are periodically refreshed by the issuer. An issuer may revoke a token by choosing not to refresh it at the next cycle. While this approach is good enough for many applications, CAPLets also optionally supports immediate revocations if needed, which requires storage at the server (as described below).

When a token holder wishes to immediately revoke a token she has shared, she sends a revocation request to the origin. The server places the tag of the token on a blacklist. Any request that contains the token with a tag in the blacklist is rejected by the server. The check is applied on every frame and therefore, revoking T implicitly revokes $\forall DD \in \mathcal{S}_{\mathcal{T}}$ as well. To prevent the unbounded growth of the blacklist, each entry carries the expiry

date of the token along with the tag. Upon expiration, the entry is removed from the blacklist since the expiry mechanism will prevent the use of the token. In the unlikely event the list grows to unacceptable levels, the owner may revoke R, which revokes every token and clears the list.

Note that the device in this case need not refresh tokens – it only maintains the blacklist – since the tokens may be (and usually are) derived by the owner on a resource-rich platform. That is, the owner may issue (and refresh) tokens that are validatable by the device without communicating with the device.

4.4.7 Request Construction & Validation

All other capability systems use the token only to authorize a request delivered alongside the capability. The server checks if the provided capability has permissions to perform the request. Because the request is included separately, it is possible (and often convenient or expedient) to transmit a capability with higher privilege than the request requires. This is arguably a violation of the Principle of Least Privilege [158].

CAPLets takes an alternative approach and unifies service requests and capabilities. We exploit the fact CAPLets capabilities are arbitrary objects, and encode requests as special capabilities. Each service request is expressed as its own capability type by placing it in a leaf frame. For instance, a `read` request on a file can be placed inside a leaf frame where the previous frame includes read access rights for the file. In this case, the file read permission is a superset of `read` request capability on that file. CAPLets' construction ensures the derivation checks succeed only if the frame before the request frame has sufficient access rights to perform the request. We denote request capabilities as C_{Req}.

A request capability must specify an operation precisely. We enforce this by man-

dating a request capability not be the superset of any other capability. In practice this means that any frame containing a request capability cannot be further derived. This ensures that once a request token is crafted, it is immutable. It can be performed as it is or discarded, but partial application is not allowed. This construction ensures the integrity of requests even when transmitted in plain text, allowing for enhanced efficiency when confidentiality is not needed.

It is sometimes convenient to create requests that depend on other tokens, for instance as proof of an earlier action such as authentication. Tokens passed alongside a request token for supporting purposes are called *auxiliary tokens*. To prevent an actor from using the auxiliary tokens separately with other, unintended, requests, CAPLets requires that clients to place a *signature constraint* with the signature of the request token in the leaf frames of each auxiliary token. This constraint ensures that a token is only valid if it is being used alongside a token with the specified signature.

Upon receiving a request, a server performs three validation steps on all tokens:

Signature verification: The server reconstructs the final tag by starting with the root tag and replaying derivations in the chain. If the resulting tag does not match the tag provided by the token, the request is rejected.

Derivation validation: Every link in the derivation chain is validated for valid derivations as described in 4.4.2. Every constraint in the previous frame must be verbatim present in the next frame. If these conditions are met in every link in the chain, the last frame is legally derived by definition. If any invalid derivation is performed at any step, the request is rejected.

Constraint validation: Constraints in all the leaf frames are checked. If any of the constraints are unmet, the request is rejected.

If the request is not rejected in any step, it is served.

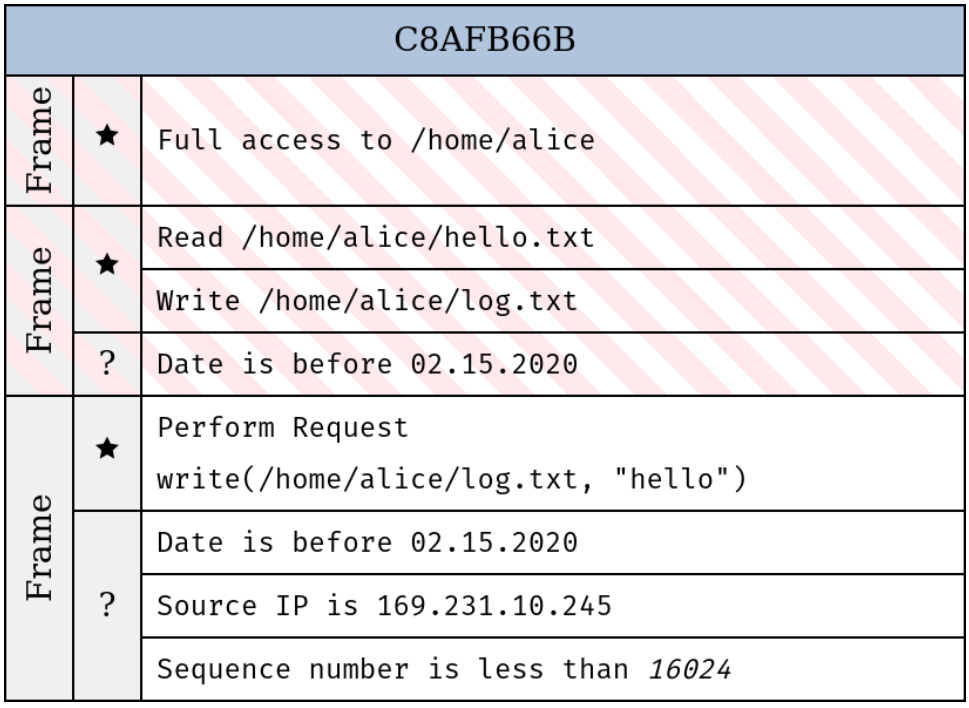

Figure 4.3: A request token comprising implied rights carried by previous frames. A request token may only have request specifications and constraints in it's last frame. Since the request is delivered as a derivation, no further checks are necessary.

4.4.8 Identities in CAPLets

To represent identities in CAPLets, we introduce *identity capabilities*. An identity capability for a certain identity, for instance Bob, is proof that the holder of such a capability is indeed, Bob. The issuance of such identity capabilities is not directly specified by CAPLets. They can be directly issued by the owner of the device, Alice, if she wishes to implement an authentication service herself. Identity capabilities use the same mechanism as regular capabilities, and thus support derivations. This means that the issuance of identity capabilities can be offloaded to a trusted identity provider, as with regular capabilities. This provider can use existing authentication services such as LDAP [142] and issue CAPLets identity tokens for successful authentications. Such tokens should have a timeout constraint to facilitate revocation. Since the tokens are generated by a computationally powerful and non-power-constrained machine, users can

routinely refresh them.

With this formulation, the user Bob can acquire a token, T_{Bob}, that proves to the device that he is Bob. Note that regardless of how Bob receives it, $T_{Bob} \in \mathcal{S}_{\mathcal{R}}$ and it can be verified with the regular derivation process, precluding the need for a complex public key management for the server. Either the owner issued the token directly, or she shared an intermediary identity root token with an identity provider, which then derived an identity token for Bob from it.

An identity constraint limits a token to be only valid if presented alongside a specific identity capability. When Alice wishes to limit a token to be only used by Bob, she places an *identity constraint* on the token and shares it with Bob. When performing a request with that token, Bob includes his identity token, which the server checks using a constraint validator. Without the identity token, the access right token will not be validated. Figure 4.4 demonstrates an end to end interaction with an external identity provider.

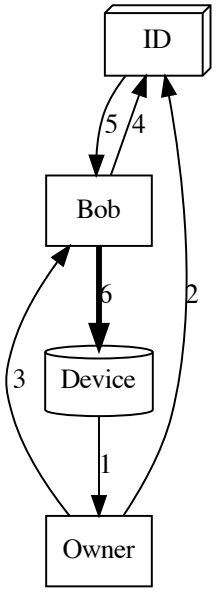

Figure 4.4: 1. Device delivers the root token to the owner during provisioning. 2. The owner registers the device with an identity provider service. 3. Owner shares a Bob-id-constrained token with Bob. 4-5. Bob receives a Bob-id token to be used with the device. 6. Bob delivers a request token alongside his id token. The thick edge denotes a client-server communication and is a CAPLets channel. The thin edges use a TLS based protocol, such as HTTPS or SSH.

An identity token is only ever used as an auxiliary to another request token and is protected by the signature constraint explained in Section 4.4.7. Note that the body of the identity capability and constraint types are dependent on the desired identification mechanism. The possibilities range from a single integer to variable length strings encoding complex identities. Since an identity token generated for a specific server will not be valid for a different server due to signature mismatches, even a single integer provides security.

4.4.9 CAPLets Key Exchange

So far we have focused on authorization. While CAPLets can securely perform request authorization without an encrypted channel due to its immutable request construction, the bodies of requests and responses are visible to attackers. Moreover, responses are not authenticated. The conventional approach to this problem is to employ TLS channels [78, 25, 42, 65]. However, TLS based encryption has many drawbacks including certificate management on devices and, as we quantify in our evaluation, requiring slow and power hungry operations. While TLS might be as efficient as possible for the guarantees it makes, we believe that not all such guarantees are as desirable in an IoT context (versus an Internet/cloud context). In this section, we describe how we exploit the cryptographic construction to provide a limited but efficient form of encrypted channel among CAPLets parties. We provide a formal proof of the security of our algoritm against passive, replay, and man in the middle attacks as an Appendix.

At the core of our algorithm is the observation that $tag(T)$ forms a shared secret among the users who hold any token in $T \cup P_T$. That is, a party can recover the tag from a body if and only if they hold a token in this set.

As described above, we use this property to ensure token integrity. However, it is also possible to use it to derive a secure symmetric key among anyone within this set, particularly, between the server and any user, since the server holds R, the root token.

For this purpose, we introduce a new leaf capability type: an encryption capability, denoted as E. An encryption capability E has the following properties: $\forall CE \subset C$ and $\forall CC \not\subset E$. This means that an encryption object can be derived from any capability and no other capability can be derived from an encryption object.

When a client wishes to set up an encrypted channel with anyone holding $T \cup P_T$, they derive an encryption token T_E from T that has only the encryption capability in the leaf

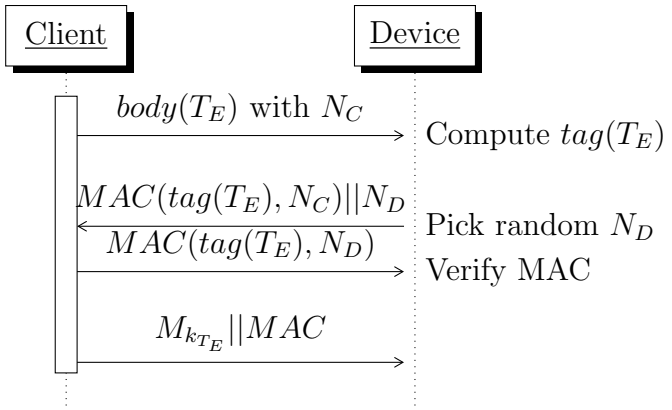

Figure 4.5: CAPLets key exchange: $tag(T_E)$ is not recoverable by an attacker and forms a shared secret. Both parties can ensure the other end *knows* the secret by sending each other challenges. Once authentication is done, the session key k_{T_E} is computed as $KDF(tag(T_E))$. M_k means the encryption of M with key k with some symmetric cipher. The MAC is computed over M_k.

frame and **transmits the token body without the tag** to the server. Upon receiving an encryption token, the server will recover the shared secret $tag(T_E)$. At this point, both parties can generate matching keys k_{T_E} for encryption through a Key Derivation Function (KDF). Both parties then challenge the other side to compute $MAC(tag(T_E), N_{\{C,D\}})$, $N_{\{C,D\}}$ is chosen at random by each party respectively.

The challenge uses the same construction we use to compute derived tags and may seem susceptible to an attack where an attacker asks a party to compute the tag of a S_{T_E}. For this reason, we have defined T_E to be non-deriveable, i.e. $S_{T_E} = \varnothing$. While the cryptographic construction allows such a token, the logical construction prevents its use. This is a case where past approaches to capability construction are insufficient to accomodate arbitrary metadata channels.

Instead of using $tag(T_E)$ directly to derive encryption keys, an authenticated Ephemeral Diffie-Hellman (DHE) key exchange can be employed to enhance this algorithm with Perfect Forward Secrecy (PFS). In DHE key exchanges, both parties generate a temporary public and private key pair and share the public parts over the network. Upon receiving the public key of the other party they combine their private key and the public key of

the other party to agree on the same key. Since any potential attacker only sees the public parts, they cannot compute the shared key. However, unauthenticated DHE is susceptible to trivial Man in the Middle (MitM) attacks. To prevent this, the DH public keys are authenticated. For instance, if both parties had long term public keys, they could sign their messages so that the other end can confirm with the long term public key.

In the case of TLS, a digital signature algorithm such as RSA or ECDSA (Elliptic Curve Digital Signature Algorithm) must be used for this purpose. However, digital signatures are compute intensive and consume significant energy. In CAPLets, the parties instead use the shared secret $tag(T_E)$ with an HMAC to authenticate their DH public keys (Q_C and Q_D in Figure 4.6).

For CAPLets, PFS means that if $tag(T_E)$ is compromised at some future time, an attacker cannot decrypt any past communication even if they stored all the packets between the parties. However, as we will demonstrate in our evaluation, PFS consumes an order of magnitude more energy than the rest of the entire communication. In practice, IoT communication often loses value with time, and by the time an attacker recovers a tag and decrpyts past data points, the data may already be obsolete. When PFS is necessary, the CAPLets implementation is still more effficient than TLS due to its ability to use MACs instead of digital signatures.

Once the key exchange is finished, the application is free to use the key with any suitable symmetric cipher. Our prototype uses AES-CTR and HMAC-SHA256 in a correct encrypt-then-MAC construction.

As mentioned above, anyone holding a token in P_{T_E} can actively intercept this key exchange. Since $tag(T_E)$ is the only shared knowledge among the device and a user, this is impossible to prevent (even with TLS, certificates would have to be pre-exchanged, i.e. another shared knowledge). However, $|P_{T_E}|$ can be minimized, for instance by the owner

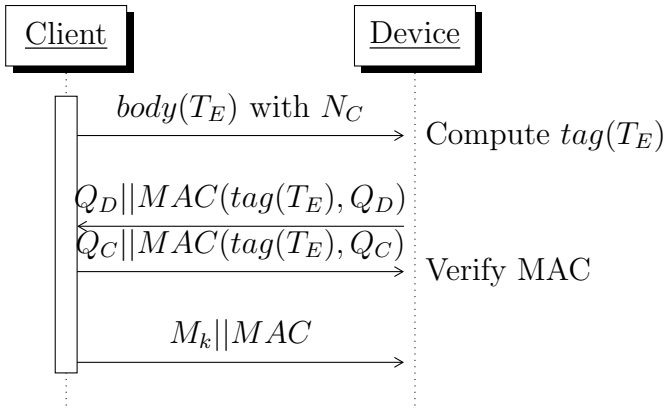

Figure 4.6: Perfect-forward-secrecy secure CAPLets key exchange: $tag(T_E)$ is not recoverable by an attacker and forms a shared secret. Both parties generate ephemeral public-private key pairs $(Q_C, d_C), (Q_D, d_D)$ and share tagged public keys. Parties authenticate each other by verifying the received MAC. After authentication, session key k is computed as $d_C * Q_D$ and $d_D * Q_C$ for the client and device, respectively. Note that the session key is not derived from $tag(T_E)$. M_k is the encryption of message M with key k via some symmetric cipher. The MAC is computed over M_k.

issuing special purpose tokens to users that are only used to derive encryption tokens. In that case, $|P_{T_E}| = 1$, with the device owner being the only other party who can listen to the conversations. We view this last point as a desirable property as a device owner should be able to monitor communication to/from the devices they own.

4.5 Special purpose authorization VMs

As mentioned previously, while CAPLets supports authortization logic to be expressed using arbitrary programs, it does not specify or rely on a particular virtual machine to do so. Theoretically, such programs can be written for the JVM, CLR, WebAssembly, BPF/eBPF, JavaScript etc. Practically, use of most of these widespread VMs in a microcontroller is impractical at best and impossible at worst for a variety of reasons including requiring a garbage collector, runtime compilation and having a large footprint.

Considering the programs for CAPLets are not supposed to be general purpose (you

should never be writing a web server in a CAPLets policy), use of a general purpose VM incurs unnecessary overhead. Plus, each application may benefit from the ability of having a purpose-built virtual machine as opposed to a general purpose, authorization focused VM. In this section, we describe the design of a customization focused VM framework called TVM that we use to build CapVM.

With TVM, we have the following goals:

- **Easy to customize**: adding custom instructions to the VM must be as easy as possible to expose unique functionality into the VMs.
- **Small footprint**: TVM environments are expected to run on microcontrollers. The generated code must be as small as possible, and consume as little memory at possible at runtime.
- **High performance**: while we would not be able to beat JIT languages for long computations, TVM programs must execute as quickly as possible to conserve battery life.
- **Easy to maintain**: use of arcane languages and tools to specify the VMs increase maintenance problems as they create new learning barriers, workspace setups and the keep-up of another project. We would like to use existing languages and tools as much as possible.

4.5.1 Design of TVM

We design TVM to allow a programmer to be able to easily specify the machine architecture (register vs stack machine), machine state (register count, processor flags etc.) and instruction implementations. A machine state is represented as a C++ structure and each instruction is implemented as a C++ function.

We use the C++ programming language to perform the actual VM generation at compile time using its excellent generic and compile time programming features. A programmer specifies each instruction available to the virtual machine using a compile time C++ list with their associated instruction identifiers (i.e. opcodes). TVM could trivially generate the identifiers itself, however, changes in the definition could change the identifiers and break existing programs. For instance, say an existing version of a TVM definition has the following instructions: `add`, `sub`, `jmp`, `print`, `call`. TVM could assign opcodes $1, 2, 3, 4, 5$ to each of these instructions respectively. However, when the programmer realizes the `print` instruction should actually be a runtime function instead of an instruction and removes it in a future version, the opcode of `call` would change since TVM cannot know about old versions of the VM. To maintain binary stability, we require programmers to assign the numbers manually to instructions so that deprecations and removal of instructions is possible.

As the VM definitions are completely declarative, the programmers do not have to have any domain expertise in virtual machine design or low level programming experience. TVM instructions are strongly typed functions. They specify what operands they take by just taking parameters of their operand types. TVM can then introspect the function arguments and automatically generate instruction decoders automatically.

For instance, the following instruction declarations specify how the actual binaries will be decoded:

```
void add(vm_state& state,
         register_id<4> a, register_id<4> b,
         register_id<4> dest);
void ijump(vm_state& state, register_id<2> indirect_register);
void reljump(vm_state& state, int_literal<16> offset);
```

`add` takes 3 registers of 4 bits each, `ijump` performs an indirect jump to the address contained in the specified register, but that register can be only of the first 4 registers, since it takes a register id of 2 bits only. `reljump` performs a relative jump, but the jump offset must be a literal of maximum 65,535 bytes.

The programmer need not concern themselves with such low level details, and prototyping and development can take place rather quickly. This could be achieved in other frameworks by asking the programmer to specify the same information twice, once in the function declaration and once more in a special `instruction_descriptor` structure. In such an approach, such declarations can go out of sync and cause hard to debug failures. As the saying goes, "A person with a watch always knows the time. A person with two watches never does.", a system with a single declaration is correct by definition, one with two declarations can never be.

After instruction decoding, TVM also generates the execution loop. Generation of an efficient execution loop is non-trivial and is a topic of research. For instance, for maximum performance on machines with sophisticated branch predictors, a threaded loop performs better than a `switch` in a loop, but performs worse on microcontrollers. Likewise, a `switch` in a loop performs better than a jump through an array of function pointers when the opcode space is sparse but the function pointers perform better when the opcodes are dense. Relieving the programmer of such concerns allows TVM to pick the most efficient strategy for each deployment.

4.6 Experimental Evaluation

In this section, we evaluate CAPLets using an end-to-end experiment that couples an IoT device at the edge with the cloud. We compare CAPLets against two mature and commercially available systems for implementing IoT applications using cloud computing.

	CC3220SF	nRF52840	ESP8266
Processor	80 MHz Cortex-M4	64 MHz Cortex-M4	80 MHz Xtensa
Memory	256KB	256KB	80KB
Crypt HW	Yes	Yes	No
Network	WiFi	BLE	WiFi
Coremark	87	212	191

Table 4.1: 32-bit devices used in the experiments.

We also analyze the performance of CAPLets using a set of microbenchmarks to help illuminate its efficiencies.

4.6.1 Devices, Software, and Setup

Table 4.1 shows the resource constrained hardware platforms that we consider as end devices in this study. We consider three 32-bit microcontrollers, each with different resource configurations (processor speed, memory size, and network type). Two of the three devices have hardware that performs cryptographic operations efficiently (demarked `Crypt HW` in the table). Coremark [62] refers to the approximate compute power of each processor.

We execute CAPLets on these devices on bare metal, using cloud SDKs and FreeRTOS from Amazon Web Services (AWS) and Microsoft Azure, and using CSPOT [183] – an operating system and runtime designed for cloud and IoT applications. For all experiments, we optimize for code size and use the same drivers across devices.

Our bare metal environment consists of a minimal implementation designed to achieve and measure the maximum efficiency of the methodology in isolation. As Macaroons [42] has no implementation that can run on a microcontroller, we benchmarked it and CAPLets on an x86 PC.

We conduct experiments using two deployments:

Edge: The device communicates with an edge system (i.e. an Intel NUC [96]) on

the same WLAN network. For AWS and Azure experiments, the device communicates with a Greengrass or IoT Edge virtual machine instance on the NUC over MQTT [31]. CAPLets communicates with the instance via TCP.

Cloud: The device communicates with resources hosted in a public cloud (connected via a fast academic network located in California). For AWS and Azure, the device communicates with an IoT Core or IoT Hub instance, respectively, over MQTT. These instances are located in Oregon and California, respectively. CAPLets uses an AWS EC2 instance in Oregon.

We configure AWS and Azure software (and reprogram the device) to use the different communication end points (edge and cloud). For CAPLets, neither the software on the device nor on the edge had to be modified since the device is oblivious to the origin and the same authorization mechanism works regardless of the client location (i.e. it uses our first-class-devices model).

We measure energy consumption by sampling the momentary power use of the processors. We use an INA219 sensor which we set to 2KHz (the maximum sampling rate the sensor supports) and the highest resolution.

4.6.2 End-to-end Evaluation

As an end-to-end experiment, we measure the time and energy costs of a sensor capturing and communicating a sample to a user. We implement this application using the services and software provided by the leading IoT service providers, AWS and Azure.

Specifically, we use Amazon FreeRTOS on the device side, AWS Greengrass on the edge, and AWS IoT Core on the cloud for AWS. For Azure, we use FreeRTOS with the Azure SDK, IoT Edge, and IoT Hub for device, edge, and cloud, respectively. Communication is handled by AWS or Azure IoT SDK libraries, which use an MQTT [31]

implementation of a publish-subscribe protocol between the end device and edge or cloud. The providers issue each device a private key and certificate to communicate with the services. We inject the key into our CC3220SF device at firmware build time. When possible, CAPLets and the SDKs perform the cryptographic operations in hardware.

We also evaluate an implementation of the application for CSPOT [183] – an experimental and open source operating system designed specifically for coupled IoT-and-cloud applications. CSPOT is useful to this study because it is a complete system capable of implementing device-level services directly, without a proxy, as well as services hosted by edge and cloud resources.

AWS and Azure implement proxy-based approaches in which an edge device or a cloud resource (but not a resource-restricted device such as a microcontroller) store access control lists for all users and devices. The resource also performs authentication.

In AWS and Azure, the device wakes up every 5 minutes, collects a sample, associates with the WiFi network, establishes a secure connection to the respective server over TLS (authorization is implicit in the establishment of this channel), sends the sample over MQTT and goes back to deep sleep. When users wishes to view the samples, they visit the cloud/edge service and read the samples.

For CAPLets, the device wakes up every 5 minutes, collects a sample, records it and goes back to deep sleep. When a user wishes to view the samples, they initiate a connection to the device. As we employ 802.11 Long Sleep Interval (LSI) feature, the device is soon woken up by the network processor, the communication succeeds, the user authorizes herself to the device, and receives the samples. This means that for *uninteresting samples*, i.e. samples that the user never reads, CAPLets conserves battery power. LSI allows a WiFi station to stay associated with the access point during prolonged sleeps. Instead of dropping received packets, the access point buffers them until the next beacon frame sent out frequently (at least once every few seconds). The

Operation	Time ms	Enrgy mJ	Code KB
No Auth Edge	95 (20)	21 (4)	45
CAPLets Edge (bare metal)	99 (28)	22 (6)	48
No Auth Edge + CSPOT	119 (12)	22 (2)	49
CAPLets Edge + CSPOT	119 (16)	24 (3)	51
AWS Greengrass	825 (231)	148 (49)	75
Azure IoT Edge	1715 (119)	251 (22)	75
CAPLets Cloud	121 (28)	26 (5)	48
CAPLets Cloud+CSPOT	165 (25)	32 (6)	51
AWS IoT Core	1696 (908)	314 (90)	230
Azure IoT Hub	3168 (244)	457 (32)	130

Table 4.2: End-to-end application performance. All execution measurements and standard deviations (in parentheses) are over 100 consecutive executions. We consider an edge (top table) and cloud deployment (bottom table).

beacon frames are handled by the low level network hardware and the device does not wake up unless there is a packet for it. Therefore, a CAPLets user will not wait for 5 minutes before they can read existing samples, but at most a few seconds.

We measure end-to-end performance and energy use from the sensor server (end device) perspective. We execute the application 100 times for each deployment (Azure, AWS, CAPLets, and CAPLets with CSPOT using edge and cloud configurations). We compute the average and standard deviation for awake time in milliseconds and energy use in millijoules. Awake time is the time it takes for the processor to wake from deep sleep and associate with WiFi network to handle one sensor sample and go back to sleep. We also measure application code size. Table 4.2 shows the results.

We include `No Auth` experiments for completeness which is CAPLets request han-

dling with security checks disabled. These results show that the CAPLets security checks only account for about 4% of the overall processing time as can be seen by comparing the No Auth row with CAPLets Edge. CAPLets code size with or without CSPOT is 4 and 2 times smaller than that of Amazon FreeRTOS with both AWS and Azure SDKs, making space for more features on the end device. The results for CSPOT show that CAPLets, when integrated with a full system that spans device, edge, and cloud tiers in an IoT deployment, adds no discernable overhead.

The latency improvement should not be considered only in the context of user-perceived latency, but also in the context of battery life. As the CAPLets application serves the request and goes back to deep sleep, TLS based versions are still busy performing the handshake. As this operation is performed for every sample for the cloud systems, it inevitably will consume its battery earlier, resulting in unavailability.

For the cloud deployment, AWS and Azure both require the devices to be awake for an order of magnitude longer per request than using CAPLets. Part of the disparity stems from the fact that our approach minimizes the responsibilities the end device has to perform per sample. Consider the steps taken *for each data point* on the AWS set up: (i) perform a DNS look up to determine the remote end point in AWS, (ii) perform time synchronization for certificate expiry verification (iii) verify the authenticity of the remote end point by walking a certificate chain (iv) establish a TLS session, (v) establish an MQTT session, and (vi) transmit the sample.

Although some steps can be cached, and *are in these experiments*, they must execute periodically on the end device. Except for MQTT, no protocol involved in these steps was designed for a memory, processor, and power constrained microcontroller. Our approach offloads costly operations to the clients. For instance, instead of the device performing a DNS lookup to locate the server, the client performs a DNS lookup to locate the device.

4.6.3 Microbenchmark Evaluation

In this section, we conduct a broad range of microbenchmark experiments to expose the performance characteristics of CAPLets. In particular, we evaluate the costs of cryptographic primitives that are performed per request for CAPLets versus competitive approaches.

In its core CAPLets depends on 2 cryptographic primitives: HMAC-SHA256 for token construction and message authentication and AES-CTR for securing the communication. If PFS key exchange is used, it also uses ECDHE for temporary session key exchange.

The primary competitors of CAPLets are TLS-based cloud services and Vanadium [65]. Both these approaches use asymmetric cryptography to authenticate parties. For IoT systems, Elliptic Curves are preferred [124, 28] due to their smaller key sizes for equivalent security [33] and better performance characteristics when compared to RSA [105]. Therefore, we only consider cipher suites with ECDSA.

For the ECDHE experiments, we use Curve25519 for its efficient implementation [37]. For the ECDSA experiments, we use the p256 [108] curve. These experiments demonstrate the cost of using long term public keys for parties to authenticate themselves. These operations are performed when a device acts as a client for cloud or edge service, as in AWS IoT Core and Greengrass.

Table 4.3 presents the average energy consumption in microjoules for each microbenchmark for the three hardware platforms. Note that timings for this set of microbenchmarks are comparatively similar showing the same ratios of performance in comparison (i.e. CAPLets is one to two orders of magnitude faster). We omit the timing comparison due to space constraints.

TLS based protocols sign ECDHE messages with ECDSA in the handshake, so their

Operation	CC3220SF	nRF52840	ESP8266
ECDHE (SW)	13679.07	1596.87	116592.71
ECDHE (HW)	N/A	537.45	N/A
AES128-CTR (SW)	73.95	9.76	96.72
AES128-CTR (HW)	19.70	1.73	N/A
HMAC-SHA256 (SW)	75.92	8.42	93.32
HMAC-SHA256 (HW)	10.65	4.94	N/A
ECDSA Sign (SW)	15592.68	2730.87	122617.24
ECDSA Sign (HW)	15337.49	456.05	N/A
ECDSA Verify (SW)	16914.24	3007.92	138659.75
ECDSA Verify (HW)	17023.39	457.65	N/A

Table 4.3: Average energy consumption for microbenchmarks over 100 runs. The units are microjoules.

Handshake	Energy Use
TLS PFS	1451.16
CAPLets PFS	547.33
CAPLets Non-PFS	9.88

Table 4.4: Energy consumption for the handshakes of TLS and CAPLets on nRF52840. The units are microjoules.

cost is ECDHE + ECDSA Sign + ECDSA Verify. With CAPLets, the cost is ECDHE + 2 * HMACs. Our results show that an HMAC uses 2 to 3 orders of magnitude less energy than either ECDSA operation. After key exchange, both TLS and CAPLets switch to an efficient, symmetric cipher.

Our efficiency gains stem from the fact that over a connection, a TLS protocol uses every operation in Table 4.3. CAPLets on the other hand eliminates the use of the bottom cluster, and uses ECDHE only for PFS. Table 4.4 shows that with PFS, a CAPLets handshake consumes about 3 times less energy compared to TLS. If PFS is not needed, CAPLets consumes at least 2 orders of magnitude less energy.

Note that the hardware-based ECDSA implementation on the CC3220SF performs on-par with or worse than the software implementation. The ECDSA "acceleration" takes place on the network processor of the CC3220SF, which has a binary-blob firmware,

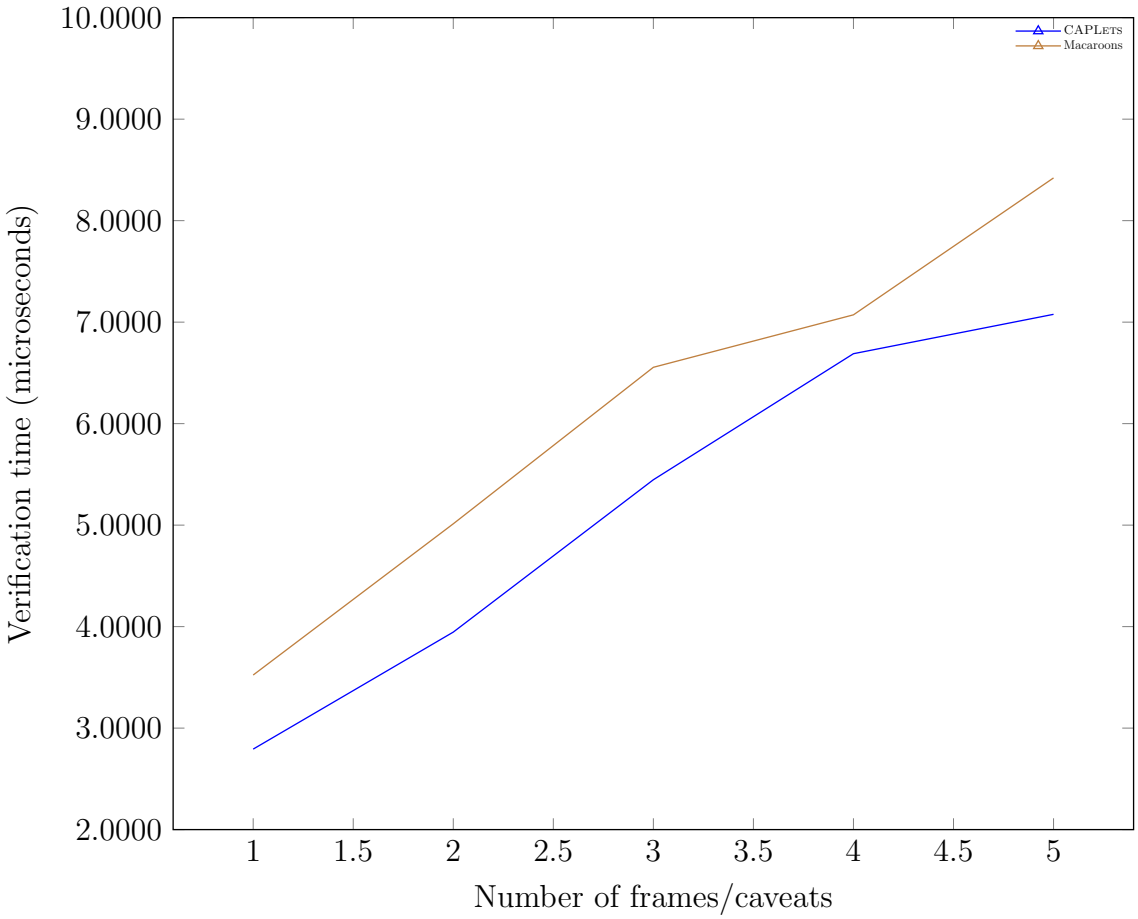

Figure 4.7: Time it takes for CAPLets and Macaroons to verify a token.

preventing us from investigating further. This observation, however, is consistent with existing literature [145]. Thus we consider it anomalous and likely due to a firmware bug or misconfiguration specific to ECDSA on the CC3220SF.

Finally, we benchmark CAPLets and Macaroons verification operations for different frame counts (using frames in CAPLets and and caveats in Macaroons). We show the results in Figure 4.7. Surprisingly, CAPLets performs slightly but consistently better than Macaroons even though the cryptographic construction is the same. To ensure there is no difference in cryptographic implementation, we modified our code to use the HMAC implementation used by Macaroons. We find that the performance difference is due to

the use of zero copy deserialization in CAPLets. Once Macaroons receives a packet over the network, it allocates memory for each caveat and deserializes its format into the buffers, which take linear time. CAPLets, alternatively, operates on the received buffer directly with no deserialization step.

4.6.4 Evaluating Constraints

We next evaluate the performance of dynamic constraints (cf Section 4.4.5). We consider three virtual machines: (i) eBPF [12], which is used in the Linux kernel for policy control; (ii) a popular VM called WebAssembly [179] (Wasm); and a low-level VM that we developed, called `CapVM`, that is specifically designed and optimized for CAPLets.

eBPF is a VM originally designed to filter network packets without involving the user space. Since the programs run in the kernel space, the runtime is well-isolated and its programs are limited in many ways (execution limits, 5 parameter limit for functions, lack of a linker etc).

Wasm is a general purpose VM designed for efficient execution in web browsers. It defines a stack machine and a memory-safe execution environment so that "unsafe" languages such as C or C++ can use it as a compilation target for execution in a browser. In this study, we use state-of-the-art eBPF and Wasm interpreters [174] and [17].

Our goal with the CapVM bytecode design is compactness and direct interpretability on embedded systems. Our design uses a register based ISA and provides complex instructions to access system resources and validation context, in addition to those for arithmetic, logical, and control operations. Its bytecode uses variable size instructions for maximum code compactness with configurable fixed size opcodes for efficient decoding. The ISA is untyped and does not perform any dynamic type or memory safety checks.

For sandboxing, all the memory accesses are confined to the fixed, linear memory of the VM, so any unsafe user code is contained within the program and cannot affect the rest of the system. Any fault during the execution results in the current request being rejected. CapVM will be released as open source if/when this paper is published.

To evaluate the performance of this bytecode and its interpretation, we use the CC3220SF device. We consider a simple timeout constraint for this study. The constraint, written in C++, renders a token invalid after a certain amount of time has passed. We compile the code to eBPF and Wasm using clang [114] with size optimizations. Because no higher level language yet exists that can emit code for CapVM, we write the constraint code by hand, directly in the CapVM bytecode language. We impose an executed instructions limit to all VMs to prevent policies from running indefinitely. In case an execution exceeds the limit, the constraint will be rejected.

We show the measurements from these two implementations of the bytecode constraint in Table 4.5. We evaluate VM Code Size (row 1), Bytecode Size (row 2), Validation Memory size (row 3), and Validation Time (row 4). Columns two to four show the results for the various alternatives. The final column (for reference) shows the metrics when we implement the constraint in native code.

This experiment shows some interesing results. CapVM is an order of magnitude faster and consumes around 1% the energy of Wasm. eBPF, on the other hand, has similar performance and energy use characteristics. However, its bytecode is large compared to CapVM, and double that of Wasm. Code sizes (of their interpreters) of CapVM and eBPF are similar and add between 1 and 7 kB to the static version. Wasm, on the other hand, takes up about 55 kB of space due to its optimization pipeline. Our results show that this VM's focus on high performance hurts its memory use and performance significantly. Since policies are one off executions rather than long running tasks, we believe Wasm is inappropriate for such use.

	CapVM	eBPF	Wasm	Static
Code Size	10.5 kB	16 kB	65 kB	9.5 kB
Bytecode	23 Bytes	400 Bytes	193 Bytes	N/A
Memory	216 Bytes	512 Bytes	30510 Bytes	16 Bytes
Time	144 μs	248 μs	10560 μs	11 μs
Energy	16 μJ	20 μJ	1125 μJ	2.5 μJ

Table 4.5: Performance of bytecode constraint implementations on Validation operation: CapVM, eBPF and Wasm. Code Size is for each VM implementation with the constraint; Bytecode size is the size of the rendered constraint. Static shows the size of a direct C++ compilation of the constraint.

The large bytecode difference between CapVM and eBPF can be explained by eBPF still being a general purpose VM while CapVM is designed specifically for the interpretation of constraint checkers in resource constrained devices. For instance, CapVM provides instructions to navigate a token, whereas eBPF must emit code to do the same. Finally, eBPF uses a fixed, 8 byte instruction encoding, bloating programs.

In conclusion, both eBPF and CapVM show sufficient performance, energy and memory use characteristics to implement CAPLets constraints. Having to generate CapVM assembly manually is tedious, and is, at present, a disadvantage. However, the smaller byte code size may warrant the inconvenience. Further, eBPF imposes some strict restrictions: functions can have up to 5 arguments, any more and the function does not compile. There also is no standard linker for it, so programs must be written in a single translation unit. Wasm has no such limitations, but its state-of-the art interpreter is large and less performant in terms of speed, energy, and size.

4.7 Summary

We present CAPLets, an efficient, secure, and flexible distributed access control mechanism that provides a uniform authorization model for a broad spectrum of resource scales and non-trivial policies. CAPLets is uniquely optimized for IoT deployments with

resource restricted devices, such as microcontrollers, as the main actors. In this setting, the optimizations it provides are substantial. We believe this consideration is key to facilitating greater and more efficient security mechanisms for IoT settings.

We design CAPLets to execute efficiently on the least-capable devices in these deployments (i.e. microcontrollers/sensors with little memory and processing power, batteries, and duty cycles), while being able to scale up for use on the most capable (cloud systems). We also define new abstractions for CAPLets that can be used to build a wide range of efficient security measures for common attack scenarios and to facilitate encrypted channels. We empirically evaluate the performance of CAPLets and compare it against state-of-the-art authorization mechanisms in use today. We find that CAPLets is an order of magnitude faster and more energy efficient than this prior work for both resource constrained devices and end-to-end IoT (i.e. sensor-edge-cloud) deployments. We also show that CAPLets performs similarly or better than Macaroons [42], while qualitatively improving portability and flexibility and introducing new features, including secure key exchange without the use of TLS.

With the combination of CAPLets's efficient construction, flexible policy definition mechanism and cheap key exchange, we address the second part of our *research directions* laid out in Chapter 1.

Chapter 5

Ambience

5.1 Introduction

Applications and systems that amalgamate heterogeneous, resource-restricted, or embedded devices with traditional resource-rich compute resources cannot use a single, "universal" approach to execute on all hardware components today. Specifically, in an IoT context, low-resource devices are programmed using special-purpose or embedded technologies that then must communicate with cloud-hosted services programmed using popular and productivity-enhancing cloud technologies. While embedded programmers miss out on the productivity of high-level systems, they often enjoy efficiencies unheard of in the cloud domain thanks to building their entire image optimized for the task. Cloud technologies are not designed and are not efficient enough for low-level systems, and relatively bespoke device programming technologies are too "low level" to support productive cloud environments. This bifurcation of systems creates reliability, maintainability, security and scalability challenges.

Microservices are a popular architecture for building scalable, distributed network services and applications [136]. The microservice architecture has seen wide adoption,

with numerous supporting infrastructure projects [5, 112, 141, 102]. Applications structured as microservices are composed of many small and "simple" services (to promote code reuse and cohesion), which are typically hosted within separate isolation domains to improve fault isolation and/or implement multi-layered trust and security policies. As a result, service requests and responses between microservices are commonly implemented using typed, Remote Procedure Call (RPC) interfaces [79, 165, 160].

Since microservices design promotes the proliferation of many different services in a single application, users and administrators of these applications often employ a container orchestration technology to implement and maintain application deployments [112, 59, 48]. Using these frameworks, developers express what their deployment should be using a declarative language, and the framework realizes it by creating new instances and decommissioning stale ones [40]. However, the microservices themselves, and any orchestration technologies that manage deployments of them, rely on general purpose, commodity operating systems [53, 175, 47, 152, 51, 180] for their execution.

A combination of Devices-as-services and CAPLets enables the development of secure, microservice based IoT applications. However, such applications must still be built on existing, commodity operating systems which have never considered hosting microservices as a first class citizen. A purely computational CSpot [182] application can indeed be moved between a Linux CSpot system and a bare metal CSpot system. However, this is a very leaky abstraction, as at the first external IO call (be it sockets, files or even just getting the current time!), the application would lose its portability.

This dependence on general purpose operating systems poses two challenges that are becoming increasingly difficult to overcome on the path of building a unified platform for heterogeneous systems. The first is that the plethora of hardware platforms (e.g. embedded IoT devices, microcontrollers, special purpose processors, edge computers, security co-processors, etc.) do not support a common set of operating system abstractions, let

alone a common operating system, either among themselves or with commodity servers. That is, while most commodity servers and virtualization environments support some form of Linux or Windows, neither of these general purpose operating systems runs on *all* devices in a distributed deployment that includes special purpose or embedded systems.

Recently, large cloud providers have adopted the serverless model in their IoT offerings as well. Serverless, or FaaS (Functions as a Service) microservices [22, 77, 52, 141] provide a programming environment where programmers write simple event driven services; the runtime platform takes over the responsibility of deployment and scaling through concurrency. AWS IoT Greengrass [19] for instance, allows developers to transparently deploy their Lambda functions to a Raspberry Pi or x86 single board computer running on the edge. However, cloud providers have yet to extend the model to microcontrollers, possibly due to efficiency challenges of doing so. The state-of-the-art is that the edge and the cloud can be programmed with a uniform "Function-as-a-Service" model, but microcontrollers must be programmed using different technologies (e.g., MQTT [31], FreeRTOS [70], and IoT SDK [97]). As a result, applications, even when adopting microservices in this context must correctly compose an increasingly vast array of disparate protocols and separately-developed technologies to achieve functionality. This software heterogeneity makes them complex to develop, brittle, hard to repurpose, and difficult to maintain.

The second challenge is achieving performance. While "scale out" (the addition of separate processors to a web service to handle additional request load) has proved economical and effective in a cloud context [72, 127], it is less effective or infeasible for deployments that include heterogeneous collections of devices and processors. Additionally, in relatively homogeneous cloud-hosted deployments, as our results presented herein indicate, the generality of commodity operating systems introduces a per-node (i.e. "scale

up") performance penalty when executing microservices.

In this paper, we describe *Ambience – a new operating system specifically designed to overcome these challenges and to support distributed applications structured as microservices in heterogeneous distributed settings that include device capabilities spanning a range of scales.* We choose microservices as our programming model as it is familiar while being amenable to building event driven, scalable, efficient systems, meeting the requirements of IoT nodes. Using microservices universally means that all functionality in an IoT deployments (including resource-restricted sensors, cameras, actuators, etc.) is represented and accessed via some microservice.

Our research with Ambience postulates that it is possible to design an operating system which is both efficient enough and high-level enough to support microservices as a universal programming and deployment model. It does so by defining optimizeable, high level abstractions for constructing and deploying microservices, which include isolation groups, coroutine-based asynchrony, typed interfaces and deployment specification, among others. These abstractions expose more information that is specific to microservice implementation and deployment, than general purpose operating system alternatives.

Ambience makes use of these abstractions not only to ease programming across heterogeneous systems, but also to introspect and automatically specialize the microservices it hosts. Rather than a single kernel image shared across all nodes of a deployment, Ambience generates individual kernel images, each specialized and optimized to run the microservices on each node in a deployment. That is, Ambience includes deployment information (typically the domain of a separate orchestration framework) that it uses to generate optimized and customized operating systems images for each device or server targeted in a deployment.

We show that these optimization features allow Ambience to achieve throughputs on the order of hundreds of thousands of requests per second across isolation domains on a

single x86 core. Furthermore, the same microservices can be transparently deployed bare metal on microcontrollers and single board computers, x86 hypervisors (KVM [170], Firecracker [7] and VirtualBox [177]) with virtio [157] support, and embedded within Linux systems (to facilitate incremental transition to Ambience), without modification.

We demonstrate Ambience's flexibility and portability using a distributed wildlife tracking application, running on a heterogeneous hardware and network setup. We evaluate its performance through detailed microbenchmarks. We also compare Ambience's key characteristics to Linux as well as Azure's IoT platform [134] Our evaluation shows that Ambience can achieve high performance and efficiency across all resource tiers. In the sections that follow, we contextualize these contributions in terms of previous and related work and through an exposition of the Ambience abstractions, automated optimizations, and deployment support.

5.2 Related Work

The popularity and prevalence of microservice-based applications have resulted in many runtime and orchestration advances that automate provisioning, scheduling, and deployment of microservices [5, 112, 113, 48]. Kubernetes [112] has received wide-spread adoption from users and service providers alike. Kubernetes requires developers to specify their entire deployment in declarative files instead of manual provisioning. This makes the creation and migration of entire multi-node clusters a trivial operation. Ambience integrates these mechanism at the operating system level and leverages a similar declarative approach for deployment specification.

Serverless computing and Functions-as-a-Service (FaaS) form a recent and compelling platform for hosting microservice applications in cloud computing systems [89, 30, 86, 141, 22]. Serverless systems provide high availability, fault tolerance, dynamic elasticity

via automated, event-driven provisioning, containerized execution, and management of the underlying infrastructure. Philosophically, Ambience shares the view of microservices (implemented using FaaS) as "omniplatform" for IoT with [184] but it goes on to illustrate that miniaturizable FaaS functionality, by itself, is not sufficiently performant in terms of memory footprint and execution efficiency. Also, the authors of [184] specify no model for device I/O – a key feature in an IoT development context. The authors of [102] exploit locality across serverless microservices to replace RPC with IPC primitives to increase throughput and lower latency. The authors conclude that there remain many individual overheads. By co-designing the entire stack for deployment and performance, Ambience eliminates a significant number of these overheads.

Ambience integrates abstractions (lightweight isolation, asynchronous interfaces between trust domains, queues, groups, etc.) and tooling (deployment IDL, compilation support, deployment/code specialization) that are also found in other systems [112, 59, 87, 171]. The authors of [166, 167] introduce the implementation of asynchronous system calls in Linux by designating pages of memory as a buffer that is polled by kernel threads. However, to achieve adequate concurrency and performance, a large number of kernel threads are required, which causes memory pressure. The io_uring [13] effort is a recent approach to implementing an alternative asynchronous (async) system call for Linux [45]. However, at the time of this writing, it does not support all system calls, and does not support kernel-to-user requests. A similar queue design is used in virtio's [157] interface where a guest operating system communicates with the host through *virtqueues*, similar to the earlier Xen [32]. While they too do not support host-to-guest requests, unlike io_uring, they use a unified pool of queue elements, so the guest can issue more work with the same amount of memory without VM-exits. Unlike these approaches, Ambience supports bidirectional asynchronous communication with low kernel resource consumption over its queue interface.

Kernel bypass systems [34, 148, 60, 100] try to eliminate kernel overheads related to network processing and context switching. Ambience, alternatively, attempts to eliminate these overheads by specializing the kernels it generates to support the user space microservices they host.

In [84, 116], the authors explore the use of memory protection units on microcontrollers to improve reliability and enable the execution of untrusted code. However, these approaches do not support server or edge class machines. Authors of [29, 184, 73] show that a lightweight serverless architecture implementation running in both Linux user space and on microcontroller systems, even without memory protection, is a viable architecture for building distributed IoT systems. Ambience is distinct from these efforts in that it is a comprehensive operating system approach that supports microservices running at all resource scales.

Unikernels [122, 44, 140] have gained traction for reducing operating system overheads by merging the kernel and the application, and by eliminating kernel protection. The motivation behind removing kernel protection is that because virtual machines implement isolation between applications, kernel corruption can only affect the application using it. However, their lack of IPC primitives prevents them from exploiting locality. Ambience supports multiple isolated services running in the same VM (i.e. group) with efficient communication among them. For deployment settings in which isolation is not desired, Ambience also supports transparent placement of services inside kernel space.

[39, 95] employ strong types to enforce isolation of user provided programs inside privileged domains. Through safe user code inside the kernel, such systems allow the dynamic introduction of efficient abstractions. However, for both systems, the type system is only available in special programming languages, and does not extend to untrusted programs written in arbitrary languages. Further, the type information is not used for performance optimizations, and mainly exists to statically enforce safety. Alternatively, [150] embeds

a JIT (Just-In-Time) compiler in the kernel to automatically and dynamically create optimized codepaths, facilitating specialization. However, performing this specialization dynamically precludes Synthesis from running on resource constrained devices. Ambience performs specialization using a variety of information available statically, making it possible to run fully optimized images across resource tiers.

5.3 Design

We define and tailor Ambience abstractions to the microservice programming style, thereby omitting many abstractions found in general purpose operating systems. In this section, we overview our key design choices, their trade-offs, and our implementation approach for Ambience.

5.3.1 Definitions & Abstractions

The primary abstraction of Ambience is a *service*. A service is a collection of procedures, each with a strongly typed interface, operating on a common, ephemeral state. The procedures act as entry points which can be concurrently executed. A *service interface* is a nominal abstract type consisting of procedure interfaces, defined in an interface definition language (IDL). Ambience includes its own IDL for generating service interfaces called *lidl*.

A *node* is an abstract entity that can host Ambience services. They can be physical machines (e.g. servers, single board computers or microcontrollers) or they can be virtual (e.g. cloud virtual machines, Linux processes, a webpage running webassembly [83], etc.). Ambience provides different levels of service on different host types since it does not have the same level of control in all physical and virtual devices. A *cluster* consists of a set of nodes and *network*s that connect those nodes.

All Ambience runtime abstractions are deployed via declarative manifests. Manifests can instantiate services, set up dependencies, describe network topologies, define groups, etc. Manifests are written in a Domain Specific Language (DSL) embedded in Python3. Ambience manifests encapsulate more information than existing declarative approaches [112, 59]. Specifically, they include service interface types and dependencies, which Ambience uses to synthesize efficient code for communication, security and more. Listing 5.1 shows a deployment manifest excerpt.

```
# Services file
instance(name="detection",
         service=tflite_detection)
instance(name="camera",
    service=dcmi_camera,
    dependencies={"frame_handler": "detection"})
export(service="camera",
       networks={"udp-internet": 4898})
# Deployment file
group(name="camera_group",
      services=["detection", "camera"])
deploy(node="camera_microcontroller",
       groups=["camera_group"])
```

Listing 5.1: A sample Ambience deployment manifest.

Ambience injects service depencencies using information within manifests during construction, thereby precluding the need by each to perform service discovery. Ambience enforces type-safety in the manifests and synthesizes code that brings up all services in the correct order and passes dependencies to each service.

5.3.2 Service Groups

Microservice design advocates for the proliferation of small, simple, isolated services. In existing systems, this service decoupling is often a binary decision: two services are either separate entities deployed in isolation, or they are part of the same service. This poses an early design challenge. Developers must make design decisions about service isolation that are difficult and costly to reverse or change once development begins, and becomes more difficult to change as development matures. Further, the performance of the resulting service mesh is not typically known until relatively late in the development lifecycle and, often, isolation design decisions must be revisited to enhance performance.

Microservice design can also pose a deployment challenge in resource restricted settings. Each isolated runtime entity (e.g. a process) consumes system resources: page tables, thread structures, kernel entries, communication costs etc. Tying the allocation of these resources to each service reduces deployment flexibility and portability. For instance, deploying two related services in different address spaces may be desirable on a cloud server but not on a microcontroller. Further, a developer may simply wish to improve performance when all services *can* run in the same trust domain, by removing the isolation boundaries.

Existing commodity operating systems do not support such flexibility directly: a program becomes a process when executed and a process is not meant to be occupied by multiple distinct programs. Moreover, each process has a non-trivial amount of global state associated with it: file descriptors, signal handlers, file system root, quotas, etc.

To overcome these challenges, Ambience eliminates all global state associated with a "process." Instead, it defines protected regions of address space that can be occupied by separate microservices. To enable this, we introduce *groups* as the unit of runtime execution and deployment.

Microservices assigned to the same group share address space and are not isolated from one another. Microservices assigned to separate groups are isolated and must communicate using fast Ambience interprocess communication (IPC) as described in Section 5.3.3. Importantly, assignment of services to groups is *not* a design-time or development-time decision with Ambience, but rather a *deployment-time decision*. That is, the developer or operational manager (in a "DevOps" [55] context) can decide what assignment of services to groups makes sense in each deployment, based on site-specific trust policies, security policies, performance requirements, etc., without code modifications or duplication.

When microservices are assigned to separate groups in a deployment, Ambience automatically incorporates IPC to facilitate communication between groups. It emits direct function calls to optimize communication within a group. Note that it is not possible to make a similar decision of whether to include two service components in the same Linux process or different Linux processes at deployment time without having two separate versions of the code: one for conjoined deployment and the other for separate deployment.

Services within a group share runtime resources: the queues as explained in Section 5.3.3, event loop and associated system threads, heap and page tables. By default, Ambience allocates a group per service. A developer is allowed to create explicit groups and include the services they wish to couple.

Note that under the Ambience group resource abstraction, services do not receive implicit resources and ambient privileges. For example, there is no global file system inherited by each group in Ambience: if a microservice requires file system access, the developer can explicitly assign a dedicated file system service to it or if two services are meant to share a file system, the developer can assign both of them to use a single file system service explicitly (either within the same group, separate groups, or in any

combination.)

This flexibility is designed to support severely resource restricted devices as well as more resource-rich servers. For example, on microcontrollers with a few kB of memory, all services in a node can be placed in a single group, eliminating most of the Ambience runtime isolation memory footprint. Key to this approach, is that services need no changes when they are assigned to the same or different groups.

5.3.3 User Space Design

Microservices require the use of event driven and asynchronous programming, whereas traditional systems mainly provide a synchronous programming environment, and the user space code is expected to implement asynchrony [118, 10] on top of synchronous abstractions provided by a kernel. Considering most kernels are themselves asynchronous and event driven internally, this back and forth introduces friction and inefficiencies.

Instead, Ambience extends the asynchronous nature of the kernel to the user space. Ambience provides and manages an event loop at the kernel level for each user space. This event loop shares code with and is almost identical to the one used inside the kernel to handle hardware events. The kernel issues user-space procedure calls directly, instead of having the user space poll and route requests. This optimization reduces the workload on each service, and provides centralized, dynamic configuration parameters such as concurrency limits and a unified tracing and observability infrastructure.

Practically, groups use bidirectional asynchronous queues for communication with the kernel. Specifically, queues implement dedicated, lock-free, single-producer-single-consumer queues for both the kernel-to-user and user-to-kernel communication. Both queues index into a per-group, shared array of queue elements. The allocation of these elements is lock-free. Lock freedom here is necessary since multiple user or kernel threads

may attempt to allocate an element concurrently. Unlike existing ring or queue based interfaces [157], Ambience allows both ends of the interface to make *and serve* requests.

Ambience does not have any system calls in the conventional sense. All core OS functionality is exposed and deployed as services that are accessed just like user provided services, meaning such services can be individually omitted or deployed inside the kernel or a user space. Such flexibility is unique to Ambience and allows deployments to configure as a unikernel, monolithic kernel, a microkernel, or an entirely new class, with no changes to the base system.

Ambience has a unique user space Application Binary Interface (ABI) in which the user space does not communicate with the kernel via platform system call capabilities, but instead, exclusively through asynchronous rings. This enables a binary program to be transparently hosted in user spaces or inside the kernel. It also allows Ambience programs to be potentially portable to other operating systems, provided the ring interface and necessary core services are re-implemented. We make use of this feature for debugging Ambience services using gdb on Linux, albeit with reduced performance, as we do not yet have such a sophisticated native debugger.

One drawback of asynchronous runtimes is that such systems [118, 117, 137, 10] are typically programmed using callbacks. Callbacks add program complexity and programmer burden as they require the creation of multiple related functions to implement a single request handler. In addition, in languages which lack garbage collection, the lifetimes of the shared variables among callbacks must be carefully managed by the programmer or risk memory corruption.

User mode threads such as fibers [107] provide a compromise between callbacks and system threads. However, fibers have sub-optimal memory requirements: each fiber must allocate and retain enough stack memory for the worst case. Practically, the worst case stack size use is not statically known, and each fiber almost assuredly over allocates its

stack. This lack of memory efficiency can cause significant memory pressure in highly concurrent settings. For example, the authors of [104] report stackful coroutines are up to 93% slower than stackless coroutines on Windows.

Stackless coroutines provide the efficiency characteristics of callbacks while providing the benefits of synchronous programming abstractions. At any time, a coroutine retains only enough memory to store its working set of local variables. The disadvantage is they require compiler support to transform the coroutines into resumable functions. However, most programming languages now support coroutines. Indeed, 11 out of the 13 most popular programming languages support them [168], with the exceptions being C and Java.

Due to their superior efficiencies, wide spread availability, and ease of programming, Ambience uses coroutines as its default programming model. It also supports fibers for compatibility, with existing libraries expecting to be able to block in a deep call stack. Our low level ring interface facilitates callback-style programming as well. We evaluate the performance fibers and coroutines in Ambience in Sec. 5.8.5.

```
struct write_job;
write_job write(bytes);


auto op = write(some_data);


co_await op; // Use coroutines
sync_wait(op); // Use threads
fiber_wait(op, this_fiber); // Use fibers
bind(op, [](auto res) {}); // Use callbacks
```

Listing 5.2: Ambience's concurrency design allows programmers to supply their own concurrency primitives, preventing the combinatorial explosion of code.

Implemented naively, using multiple concurrency models together requires the same functionality to be duplicated. To avoid this, Ambience decouples concurrency primitives from units of asynchronous work, called jobs. In Ambience, most asynchronous operations set up the necessary state to realize a task, without starting it. Work starts once a completion handler is provided to the job. The completion handler can be a callback, coroutine, or fiber resumer. This allows us to implement 1 asynchronous API (for instance, a network packet transmission), and allow users of the API to bring their own concurrency model. Figure 5.2 demonstrates the developer facing API. Switching to this design from the naive version reduced Ambience's driver sizes by a factor of 3 while adding more functionality.

Ambience also provides a library of high level concurrency algorithms, e.g. `when_all`, which takes a number of jobs and creates a new one that runs when all jobs finish. We implement this with direct OS support, which allows user groups to remain unscheduled until *all* requests complete.

5.3.4 Immutability and Specialization

Microservice deployment orchestration frameworks often employ immutable and declarative languages to describe a deployment [112, 59]. These frameworks use this specification to install and start services across the nodes in a deployment. In practice, these frameworks deploy services as Linux containers. When a change is made to a service, the platform kills the old containers and starts new ones with the updated image: the containers are immutable.

Ambience employs and makes novel use of this declarative model. It generates a potentially unique kernel image for each node in a deployment, optimized for the microservice workload the node will run. Specifically, it carries relevant type and deploy-

ment information specified in an IDL and manifest files that are used for optimizations at runtime.

Ambience also supports preallocation of resources for services at build time, to reduce cold start times and to detect insufficient resources ahead of time. Currently, Ambience can preallocate the following for each group: isolation structures, system thread stacks and control blocks, queues, in-kernel group descriptors. For importers and exporters, it also preallocates networking structures, for instance UDP control blocks. When possible, these resources are initialized at compile time using `constexpr` [50] data structures and algorithms. `constexpr` allows some stateless C++ code to execute at compile time. Ambience also supports dynamic provisioning these resources: pages can still be allocated and mapped dynamically, threads can be created and destroyed, and sockets can be created at runtime, only with higher runtime cost and the possibility of failure.

5.3.5 Use of Interface Types

Microservices in Ambience communicate over statically typed interfaces defined in an IDL. This information is used to reduce load on developers by auto-generating serialization code and performing compile time checks. In commodity operating systems, these types are erased at the system call boundary by serialization: all the operating system sees are bytes coming in and out.

In contrast, Ambience maintains interface type information for as long as possible. Service interfaces are typed at deployment time and this type information is available and used when the kernel is constructed. Type information is only lost when an Ambience invocation crosses a network boundary.

Ambience makes extensive use of this information to implement efficient communication, enable compiler optimizations, gain observability, and to introduce additional

functionality. Ambience queues are strongly typed: when a user space makes a request, it does not perform serialization. Instead, it packs pointers to its arguments in a zero copy data structure generated by the IDL. As far as the user space is concerned, all Ambience requests are zero copy. If the request is to be handled in-kernel, nothing more is done. If the request is to be handled off-node, Ambience performs serialization and communication. If the request is to be handled by another user space service in the same node, Ambience synthesises specialized code using the static type information to implement efficient parameter passing. Scalar types and small vectors (strings, buffers etc.) are passed by value, and large vectors are passed by read-only memory mappings. With this design, user spaces need not manage complicated shared memory objects and buffers. The kernel manages mappings automatically since it knows what the user space needs through type information.

Finally, Ambience uses this information to synthesize a broad range of higher level functionality. For instance, Ambience provides structured logging of requests, since it can make sense of the bytes it receives, e.g., it auto-generates externally accessible REST [153] end-points to be consumed by web applications, and automatically injects sophisticated authorization code by inspecting parameters for correctness.

5.3.6 Networks

Ambience services that communicate over a network are not exposed to networking details. Instead, Ambience automatically manages networking using node and network manifests provided by developers. Moving the responsibility of network management to the operating system simplifies service development and increases portability.

Ambience's control plane constructs a graph from manifests, where networks and nodes are vertices and *importer*s and *exporter*s are edges between networks and nodes.

The edges are directed and weighted to allow for potential asymmetry in the connectivity (e.g. firewall rules that control connectivity) and allow user feedback on forwarding paths (e.g. by incentivizing a particular network or interface).

Specifically, Ambience constructs a high level communication overlay at the application layer (i.e. in terms of typed service requests rather than routing of untyped packets) rather than at the network layer as found on existing systems [112]. Working at the application layer allows it to forward requests between services automatically across a wide range of networks to achieve full connectivity for a deployed service mesh. That is, it automatically and transparently joins public and private IP networks, low-power networks such as XBee, as well as point-to-point links such as USB, SPI and UART.

5.3.7 Automatic Access Control

In typical microservice applications, most routes are publicly accessible and services themselves are expected to implement access control via a selection of authorization mechanisms such as role- or attribute-based access control, access control lists, or decentralized, token-based authorization primitives [1, 41]. This approach introduces redundant work, precludes a separation of concerns, and limits Ambience's ability to specialize services within the same trust domain.

Typically, access control implemented in microservices takes the form of sanitization: they answer the question, "Can the current subject call this procedure with these arguments?". In conventional systems, there is no other entity to perform this check other than the service itself, since no other component can make sense of the messages until they arrive at the service. Ambience, however, has visibility into individual service interfaces, and can synthesize access control code where needed. All Ambience needs is a formal specification of predicates to ensure per procedure.

Synthesized access control code can be deployed inside the service's group. It can also be used inside the kernel to avoid unneccessary context switches, and it can be run on another machine to early fail bad requests. If an Ambience service actually receives a request, it is guaranteed that access control has been performed on the given arguments.

To implement this support, we leverage CAPLets [27], an open source, capability-based authorization framework that runs on both microcontrollers and resource-rich machines. CAPLets requires policies to be defined as capabilities and constraints and written manually by developers. Ambience extends this approach to automatically generate capabilities and constraints from manifests, precluding the possibility of definition mismatch and reducing programmer burden. For requests that take place on the same machine, we employ CAPLets' policy mechanism between groups. For requests that take place across a network, we automatically inject CAPLets' network protocol, which serializes the request, signs it, adds replay protection, and optionally encrypts it. Once received by the destination node, the same policy mechanism is invoked after passing signature checks.

The keys for network requests inside the deployment are automatically managed by Ambience with no user involvement. For externally made requests, Ambience generates capability tokens that can be shared with other parties. These tokens are evaluated at explicit ingress services, which again is automatically synthesized by Ambience.

This deployment-aware access control improves the pace of development by relieving programmers from implementing access control, reducing bugs through automatic synthesis of code, and improving runtime efficiency by optimizing away unneccessary checks. It also simplifies administration by providing a uniform authorization infrastructure at the operating system level.

Case Study: DeathStarBench To demonstrate the fit of Ambience's design for real world microservice applications, we investigated the DeathStar microservices bench-

mark [71]. Given the assumption that the program is the unit of deployment, their microservices are not easily decoupled from the test harness which implements the networking, dependency, and platform configuration. Thus, we use the suite to validate Ambience's design and not for performance comparisons. The microservices within the benchmark suite all communicate with each other not through unstructured pipes or sockets, but over strongly typed interfaces, either via gRPC [79] or Thrift [165]. Ambience extends the use of these interfaces to drive optimizations across address spaces. Across the suite, we find that none of the 33 services make direct use of system calls (only the test harness uses `signal` and `exit`). This finding, coupled with our experience with microservice applications, leads us to believe that POSIX system call compliance is not a requirement in this domain and thus we omit it from the Ambience to streamline its design and specialize its implementation. To support system calls made by terminal services, such as databases, we port these services to Ambience by replacing their system calls with Ambience interfaces.

5.4 Deployment Overview

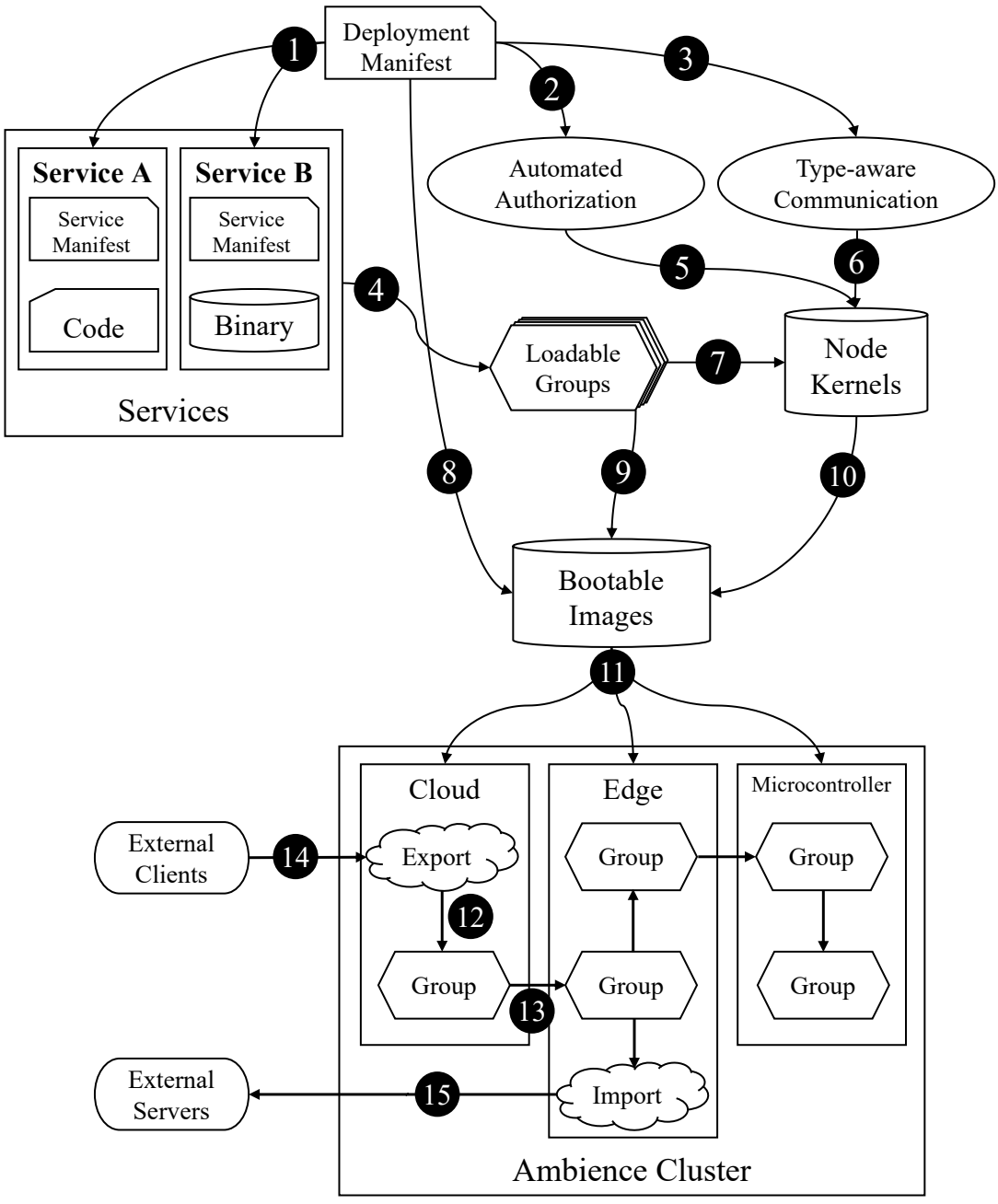

Figure 5.1: End to end overview of an Ambience deployment.

Figure 5.1 presents an end to end overview of an Ambience deployment.

1 The Deployment Manifest node is the top level manifest provided by the cluster owner. First, it specifies the nodes in the cluster and the networks. Second, it specifies what services should be included as part of the deployment. Service manifests specify the interface type of the service, as well as the interface types of all of its dependencies. The types are described using a formal interface description language that Ambience can understand at deployment time.

2 & 3 Using the type information associated with the service interfaces, Ambience automatically synthesizes code to perform authorization and optimal communication.

4 Each service manifest the artifact associated with the service. The artifacts can be ahead of time compiled binaries, or source code. Ambience creates loadable groups from the services by linking, and if necessary compiling the service artifacts.

5 & 6 Automatically synthesized programs are linked into each kernel image.

7 Metadata associated with each loadable group are linked into their host kernel image. This metadata is used to pre-allocate certain runtime resources, such as page tables and sockets.

8 & 9 & 10 Ambience bundles each kernel and group artifacts associated with a specific node in bootable images. For x86 servers, Ambience generates a bootable ISO, for microcontrollers, it generates a loadable image. Using the deployment manifest, Ambience also generates memory layouts for all groups. While a global memory layout is not necessary on hardware with paged virtual memory, microcontrollers operate directly on physical memory and each group must be loaded at a suitable location. At this stage, Ambience has all the information necessary to allocate regions and finally links each group to their actual runtime addresses. This relieves the microcontroller kernels from performing runtime relocations as well as avoiding use of position independent code to

achieve maximum performance.

11 Bootable images generated by Ambience are delivered to their nodes through a machine-dependent manner. For instance, by uploading the image to a cloud image registry for running on cloud virtual machines or by flashing the image to a microcontroller's ROM.

12 Services running inside groups on the same node communicate over Ambience's optimal, code synthesized, local Inter Service Communication mechanism. When communication occurs over local-ISC, Ambience passes arguments and return values between address spaces using the optimal strategy. It makes use of transparent memory mapping to deliver large amounts of data across protection domains with very small overhead.

13 Ambience automatically synthesizes serialization and deserialization code for communication taking place across the network as well as access control code for protecting these interfaces. The programmers do not have to write any code to support this networking infrastructure, and as far as they are concerned, all calls to them appear identical.

14 Deployment Manifests can specify explicit *Exports* to make a service available outside the cluster. Exports can specify network protocols other than Ambience's internal protocol to interoperate with existing clients. For instance, Ambience supports exporting any service over an HTTP REST endpoint with no involvement from the programmers.

15 Similar to Exports, Ambience manifests can also specify explicit *Imports* to receive services that are running external to the cluster. Imports can run over protocols other than Ambience's to provide compatibility with existing services. However, the services must provide an RPC style interface, since programmers will consume them as if they are Ambience services. For instance, streaming APIs cannot be transparently imported and may require manual programmer effort to use within Ambience, same as existing operating systems.

5.5 Memory Management

Unlike unikernels, Ambience supports multiple address spaces natively. Services running in isolated address spaces cannot communicate via direct function calls and must involve the kernel to facilitate passing and returning of necessary information between address spaces. Such communication is termed Inter-Process Communication (IPC).

Ambience's memory management system is designed with support for low end microcontroller hardware (which we refer generally as Memory Protection Units (MPU)) in mind as well as supporting full-featured MMU hardware. The main difference between MMUs and MPUs is that MMUs provide paged virtual addressing whereas MPUs only provide memory protection on physical memory segments. Lack of virtual addressing on MPUs mean that Ambience's design must embrace the ability to work under a single address space.

5.5.1 Memory Sharing

When Ambience provides transparent access to a service in another address space, the kernel will automatically map memory segments from the caller's address space to the callee's address space usually in a read-only fashion to achieve zero-copy calls.

Arbitrarily mapping pages across address spaces is a novel design of Ambience. On existing systems, shared memory requires significant coordination across the parties of the said memory. On Posix, memory can only be shared using `MAP_SHARE`d anonymous pages across a `fork` or using a shared file or shared memory objects. In all cases, programmers of both the caller and the callee must explicitly setup the sharing and make sure all arguments are in the shared area. Even the most efficient microkernels [93, 109] have to resort to copying for large buffers.

While the traditional design simplifies memory management in the kernel, it prohibits

the implementation of a true zero-copy transparent IPC mechanism.

5.6 Efficient IPC

Ambience's efficient IPC is implemented by combining the highly flexible memory management facilities described previously with the deep knowledge of what user spaces do. Using traditional systems, processes can share information through allocating a large buffer, enough to fit all parameters, copying all parameters to the buffer and passing the buffer to the other process using IPC.

Ambience programs, on the other hand, merely set up a well-typed structure containing scalars directly and pointers to large objects and passes a pointer to this object to the kernel. As the kernel has a priori knowledge about the contents of the structure, it replicates the structure on the receiving side by copying small scalars and directly memory mapping the large buffers. The sender and the receiver are completely oblivious to this sharing. Both sides see a normal, directly usable structure. Contrast this to usual systems where the sender has to serialize a message and the receiver has to deserialize it into a useable structure.

Since large objects are passed by pointer, the kernel will follow the pointers and ensure the same data structure is replicated on the other side. Whether a zero-copy transfer for a pointer is possible or not depends on the pointee's is alignment and size. For instance, if a user space attempts to share a string containing 100 characters, on a paged system with 4k pages, it is impossible to directly map the page. However, if the string is big enough and has well-aligned sections, parts of it will be memory-mapped. In the best case, no copying needs to be performed.

Ambience supports partial-copy sharing on MMU-capable systems as well. For instance, consider a string of size 8193 bytes, starting at address $4096 * k + 4095$ for some

constant k. This means that except for 1 byte at the beginning, the whole string can be mapped directly to the other address space. In this case, Ambience will allocate an anonymous page, copy the single byte to the end and map this at address $4096 * k$ in the destination address space. The rest of the data will be mapped directly at $4096 * (k + 1)$. This ensures no unintended data leaks to the destination service while using the optimal strategy as much as possible.

However, on MPU systems, partial copying is not possible if a pointee is not well aligned (and well sized!), and a total copy has to be made since it is impossible to supply different physical memory for the beginning of the string without virtual memory support. Zero-copy is still supported for buffers that are well aligned and well sized. However, as MPU hardware are usually extremely limited, the microcontroller implementation has to fall-back to total-copy frequently.

5.6.1 Implementation Details

Ambience's IPC mechanism is implemented within the kernel and thus is implemented in C++. Since the approach is fully type based, we make significant use of C++ templates to synthesize the necessary functions.

Each type that is allowed to pass through the IPC interface needs to opt-in by providing a specialization of the primary `tos::quik::sharer<T>` template. The sharer interface consists of two required static functions: `size_t compute_size(const T& arg);` that returns how many bytes of extra data would `arg` need in the destination address space. For instance, for small scalars, this function always returns 0 since such scalars are always stored within the structure itself. For a string, it would be the size of the string if a total-copy needs to be made, or 0 if the string can be memory mapped.

The next function is `T do_share(Share auto& share, const T& arg);`. This func-

tion performs the actual share. Notice that if `T` is a pointer, `do_share` returns a pointer as well.

Using these specializations, passing an entire struct can be achieved by copying the struct verbatim to the destination address space and transforming each member through `tos::quik::sharer<T>::do_share`. The resulting code for sharers is very concise, and the overall sharing code is easy to read and maintain.

Optionally, types can also opt-in to provide another function in their specializations with the signature `void finalize(Share auto& share, T& original, T& copy);`, in which the types can provide a finalization step, used for read+write operations. For instance, while the backing store of a `message_builder` is memory mapped between two address spaces, how much data is written to that storage by the callee still needs to be transferred back to the caller. Finalization handles this final metadata transfer.

As everything is implemented using static polymorphism, the compiler is able to optimize away complicated, multi-step shares to a single `memcpy` and even to a single SIMD copy instruction for smaller parameter packs.

Our memory subsystem generalizes the concept of a page in MMU systems to an *address space fragment*. An address space fragment is a range of memory in one address space that can be zero-copy shared with another address space. Naturally, fragments are hardware dependent. On paged systems, a fragment maps to a page directly and statically. On an MPU system, however, the fragments are dynamic. The rationale behind this design stems from the limitations of MPU hardware. Typically, MPU hardware requires the base addresses of the segments it protects to be aligned to the size of the segment. For instance, if a segment is 128 bytes, the base address must be aligned to 128 bytes. Therefore, there is no single fragment layout in such systems.

5.7 Lidl, the little IDL

Microservice frameworks facilitate communication between services through strongly-typed interfaces. Since microservice architecture aims to provide freedom for the implementation language of services, such interfaces are described using a stand alone interface description language (IDL) instead of using the features of a single programming language. For example, gRPC [79] uses Protocol Buffers [149] to describe its interfaces, and COM [49] uses MIDL [133]. Some frameworks come combined with their IDLs, for instance Cap'nProto [160] and Thrift [165]. These frameworks are predominantly Remote Procedure Call (RPC) protocols, and provide client and server stubs that automatically serialize and deserialize network buffers into useful objects. This indiscriminate serialize-deserialize approach makes applications topology-agnostic without any recompilation or linking. For instance, a client accesses a server through the same messaging transport regardless if the server is on another machine, another process or even within the same process.

However, this careless serialize-deserialize approach is only optimal for the remote-machine topologies. If the client and the server is in the same machine, a shared memory approach would be much more efficient, and if they are in the same address space, well it could be a direct function call. There are some RPC frameworks focused on IPC efficiency. COM [49] and FIDL [68] claim to be such frameworks. FIDL focuses on IPC as it has to manage capabilities, but at the time of this writing, it does not make use of the shared memory and still serializes data across address spaces. It also is an IPC-only framework, it does not work across a network in its current form.

In short, almost every IDL is focused on facilitating RPCs across a network, and always serializes. The handful IPC focused RPCs still perform serialization, and only exploit the IPC advantages to implement local authorization purposes. Lidl is an effort

to provide an interface description language that is optimal for every use case. Lidl interfaces can be transparently implemented by direct function calls, efficient zero-copy shared memory for cross-address space calls or serialization for cross-machine calls.

5.7.1 View Types

To achieve efficiency in almost all situations, we have to rethink how service interfaces are designed. Consider the case of passing potentially large buffers of data to/from a service. For instance, a `write` file system API. What is the type for the buffer parameter?

If we were writing generic code in C++, the `write` function would ideally be generic over the buffer type so that the caller can supply whatever they have without any conversions. However, since interfaces are exposed over virtual functions, we cannot have templates as these interfaces. We need some type erasure. Let's consider using `std::vector<std::byte>` as our byte buffer type. A `std::vector<std::byte>` is an owning array of bytes and it has a very strict invariant: it has to call `delete[]` on the pointer that it internally holds. Since it will unconditionally delete the buffer, it has to ensure the buffer it points to is allocated properly, and its constructor will always allocate buffer itself. This means that even if the caller already has a (potentially heap allocated) buffer, another allocation must be performed and all the data has to be copied to the new, managed buffer.

View types are an effort to alleviate these problems. An instance of a view type is a non-owning reference onto an existing object. In one sense, view types are the generalization of non-owning, non-null pointers.

While programming languages have started providing support for view types, RPC frameworks still specify their interfaces using either owning types, or framework specific types, usually resulting in unnecessary copies. For instance gRPC [79] uses `std::string`s

Concrete Type	View Type
`T`	`T*`
`T[N]`	`span<T>`
`std::vector<T>`	`span<T>`
`std::array<T, N>`	`span<T>`
`std::string`	`string_view`
`const char[N]`	`string_view`

Table 5.1: Common concrete types and their associated view types. Notice the view types efficiently erase away the concrete type.

as its string representation and FlatBuffers [75] uses its own string type, necessitating copies even if the callee is not a remote machine. It is also unclear how view types could be supported in these existing RPC frameworks, as their main interface is about building messages or serializeable objects.

To achieve efficiency within an address space, we designed Lidl with support for view types in mind. Lidl procedures can have view types as parameter and return types, which are reflected as correct view types, if available, in the target language. When the caller and the callee is in the same address space, a view type will do the right thing by doing nothing. A `string_view` argument will be directly referring to the string passed by the caller. If the caller and callee are in separate address spaces, the kernel will automatically make the `string_view` available on the destination address space since it has knowledge about what a `string_view` is. Finally, if the destination is on another machine, Ambience will create a gather buffer[1] from the `string_view` directly, achieving true zero-copy in all cases.

[1]Ambience's network interfaces implement scatter-gather APIs, meaning that clients can specify multiple disjoint buffers to be sent as a single packet. True zero-copy operation needs driver support as well, and Ambience's virtio-net driver implements support for it.

5.7.2 Wire Format & View Types

While Lidl is primarily used to define service interfaces, we make use of it to define a wire format to be used for inter-node communications as well.

Lidl's wire format is rather simplistic compared to [79, 75, 160, 165, 68], with the caveat of supporting view types. Each lidl value structure is mapped to a C-layout compatible `struct`, and each lidl value union is mapped to a tagged C-layout compatible `union`, which allows user programs to access lidl types as efficiently as they access regular types. Value types, however, cannot contain any dynamically sized data. For instance, integers, and fixed sized arrays of integers are value types, but a variable length string is not. Lidl implements variable sized data using *reference types*, which are accessed through relative offsets. Access through offsets is not as efficient as access via a direct pointer, but having relative offsets rather than absolute pointers allows lidl messages to be copied efficiently without any need to adjust pointers etc.

Each view type supported by lidl also has an associated wire type. If a view type is being serialized, for instance at a network RPC boundary, lidl's runtime will automatically convert the view type object into the wire type object. At the other end, a view type can be created from the zero copy message directly.

For example, consider an interface taking a `string_view` as a parameter that is actually implemented on a remote machine. At the network boundary, lidl will automatically create a `lidl::string` from the argument inside the message and transmit it to the other machine. Once received, all the arguments, including the string are in a zero-copy buffer in the memory. The wire layout of `lidl::string` is compatible with `string_view`, meaning that the string does not need to be copied out of the buffer, and can be used by the server directly. In contrast, `grpc,capnproto` would have to make a copy of this string before passing it on to the server, since they do not have a concept of view types,

and their interfaces are defined in terms of owning types such as `std::string`.

5.7.3 Network & Zero Copy Interface

As mentioned before, all Ambience communication, networked or on the same node, happens over typed RPCs. Networking is handled implicitly. If a dependency is not in the same address space as the caller, the dependency will be provided using a stub. The stub captures all arguments in a structure, that either holds each argument by value (for scalars and small vectors), or by pointer (for large vectors). The exact definition of this data structure depends on the interface types and platform, and since it holds pointers, it cannot be just passed between address spaces and machines directly. The stub also allocates space for the return value on the stack. It then passes the procedure identifier (a unique number per interface), a pointer to the argument structure and a pointer to the return value to its *zero copy backend*. Lidl stubs are class templates parameterized over the zero copy backend type. Listing 5.3 shows an example stub generated by the compiler.

```
template<class Backend>
class alarm::async_zerocopy_client final
    : public alarm::async_server
    , private Backend {
public:
    template<class... BaseParams>
    explicit constexpr async_zerocopy_client(BaseParams&&... params)
        : Backend(std::forward<BaseParams>(params)...) {
    }
    Task<bool> sleep_for(milliseconds dur) override {
        auto params_tuple_ = ::lidl::make_params_tuple(dur);
        using ret_t = bool;
        std::aligned_storage_t<sizeof(ret_t), alignof(ret_t)> return_;
        [[maybe_unused]] auto result_ =
            co_await Backend::execute(std::integral_constant<int, 0>{},
                                      &params_tuple_,
                                      reinterpret_cast<ret_t*>(&return_));
        co_return *reinterpret_cast<ret_t*>(&return_);
    }
};
```

Listing 5.3: A sample zero-copy stub client generated by lidl.

The backend can either be a proxy, in which it merely passes through its arguments to the next backend (which itself may be another proxy), or it can be a terminal, where the request will be handled. Proxy backends need not know about the types involved, they

can effectively treat the pointers as `[const] void*`. The terminal backends, however, must have a priori knowledge about the incoming types, since the types involved are not self describing. This is rather trivial on Ambience, since the terminal backends are automatically generated. Lidl provides library functions to automatically execute a procedure call on a given service implementation, given the procedure id, pointer to arguments and pointer to the return value.

In the case of a client and server running on the same machine but different address spaces, the following events will occur:

1. The caller service calls the stub implementation in the user space,
2. The stub implementation passes the arguments to the upcall backend,
3. The upcall backend places the index and pointers into the asynchronous ring and suspends the caller,
4. Immediately or later, a user mode to kernel switch,
5. The kernel iterates over the ring and executes each request element,
6. If the callee is in the kernel, the request is served
7. If the callee is in another user space,
   (a) A proxy within the kernel receives the parameters,
   (b) The proxy knows about the types, and executes the type-aware fast IPC code to pass the argument structure to the target address space, creating valid pointers in the destination address space,
   (c) The proxy places the new pointers in the destination group's ring,
   (d) Immediately or later, a kernel to user mode switch to the destination group,

(e) The Ambience runtime iterates over the ring and executes each request element,

(f) The request is handled by the actual service

8. If the callee is in another machine,

(a) A network proxy within the kernel receives the parameters,

(b) The proxy invokes lidl's serialization routine using the given argument struct (the kernel can dereference user space pointers),

(c) Since the proxy is implemented by a network device, it sends the serialized buffer to the other machine

(d) A proxy running on the destination machine receives the buffer, and after validation, treats it as a typed lidl object,

(e) The proxy executes the request, which will start from Item 6,

Hence, Ambience provides a high level, but efficient and versatile indirection interface that can be transparently used to implement many communication strategies, including fast zero copy transfers across address spaces as well as network transfers over IP, low power radios, serial ports etc.

5.8 Evaluation

We evaluate Ambience in terms of its ability to deploy end-to-end microservice meshes in different configurations without code modifications, and also in terms of its performance, particularly with respect to microcontroller support. For the former, we have developed an motion-triggered "camera trap" application used in wildlife monitoring

settings that captures a digital images from a remote camera, processes them and performs classification, and stores the results in a data repository. We use equivalent implementations for Ambience and the IoT software framework from Azure and report both quantitative and qualitative productivity metrics associated with deploying each version in different configurations.

To motivate the Ambience design decisions from a performance perspective, we then use a set of microbenchmarks to provide isolated measurements of specific functionality. We use other services than the wildlife-tracking application in our microbenchmarks to expose the characteristics of different deployments, such as the effect of service call depths. We focus on energy use, latency, portability, and scalability. In remote IoT settings, sensor and actuator nodes often use battery power (recharged during daylight hours using solar power) and operate on a duty cycle consisting of active periods and periods of low-power dormancy. The minimum duration of the active periods is defined by execution speed and communication delay. Thus power consumption is often correlated with execution duration and, hence, reduced execution duration implies less energy consumption and the use of smaller batteries, a smaller solar array, more active periods per unit time, etc., for the same communication duration. Latency measures the duration of a specific operation or set of operations, and scalability plots the performance of a node as a function of the load it hosts.

We perform the experiments on the following hardware:

Motion: nRF52840s microcontroller with an ARM Cortex-M4 core running at 64MHz, 256KB of RAM and 1MB of flash memory, a motion sensor [85], and a radio [185]

Camera: STM32F746 microcontroller with an ARM Cortex-M7 core running at 216MHz, 512KB of RAM, and 1MB of flash memory, an OV5640 CMOS image sensor [144], a motion sensor [85], a radio [185] and a 100-Mbit Ethernet,

Edge: A single core x86_64 virtual machine with 1GB of RAM running under QEMU-

KVM on Linux Kernel 5.15.6 on an AMD 5950x processor running at 3.4GHz,

Cloud: Two DigitalOcean [57] single core VMs with 1GB of RAM on Intel Skylake processors.

We run the Ambience prototype natively on **Motion** and **Camera**. Virtualized Ambience runs on the hypervisor using virtio devices with custom drivers. Unless otherwise specified, we execute the microbenchmarks on **Edge**. All code is implemented using C++.

5.8.1 Wildlife Monitoring Application

As a motivating application and to demonstrate the flexibility that Ambience makes possible, we describe an end-to-end wildlife monitoring system designed for off-the-grid locations (e.g. research reserves). Physical sensors and cameras employ embedded microcontrollers. The application uses a mobile version of Tensorflow [172] to process images either on-camera, or off-camera (possibly traversing a network link in the process) on an x86_64 edge server device and posts the analysis results to the cloud over slow, cellular internet.

In this application, the motion detector nodes run completely on batteries, making battery life paramount. The camera nodes have solar power, but power usage is still important since the camera uses a battery during nighttime operation that is recharged during daylight hours. In the deployment we use, the edge servers also use batteries, but they are from a large battery complex with a large solar array located in an open space. The cameras communicate with the edge server via an Ethernet network, and the sensors communicate with the camera via low-power Xbee radios.

We implement and deploy the following services:

Motion Sensor handles low level hardware events from the PIR motion sensors and

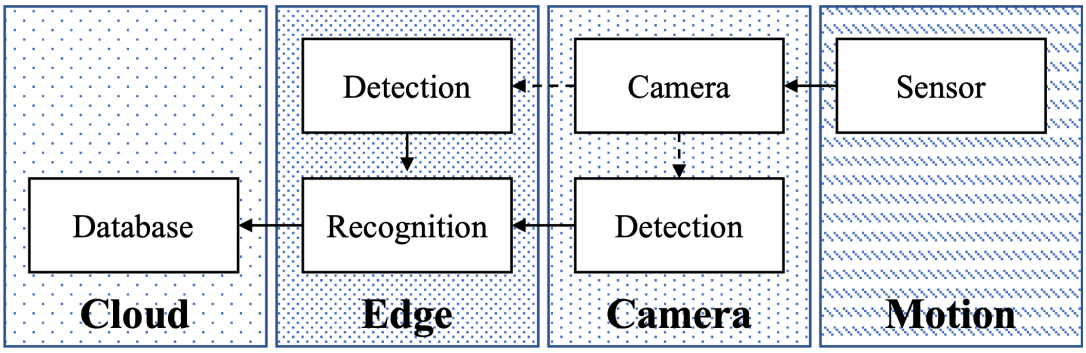

Figure 5.2: Physical service architecture of the deployment. For brevity, we omit the lower-level services such as logging, timers etc.

forwards them to its event handler. Implemented in 56 lines.

Camera manages the OV5640 camera using the STM32 DCMI peripheral, capturing a full sized snapshot everytime it is called and passes the image data to the frame handler. Implemented in 42 lines.

Detection implements an animal detection service using Tensorflow on a sub-sampled image. If an animal is detected, the frame is passed to the recognition service. The model used in this service takes up around 320KB and is fully portable across the **Camera** and **Edge** nodes. Implemented in 94 lines.

Recognition implements an animal recognition service, using Tensorflow but on a higher resolution version of the frame, and classifies the subject. The classification result is passed to the database service. Implemented in 90 lines.

Database implements an append only log of classification events. Implemented in 120 lines.

The most distributed deployment sites the motion detector at the likely entrance of a "stage" for the wildlife (e.g. a watering location) that is imaged by the camera. Thus it spans four tiers. The motion detector (tier 1) communicates with the camera (tier 2) via Xbee low power radio to trigger an image capture. The camera then communicates with the edge server which has a public internet connection. This deployment, interfacing with

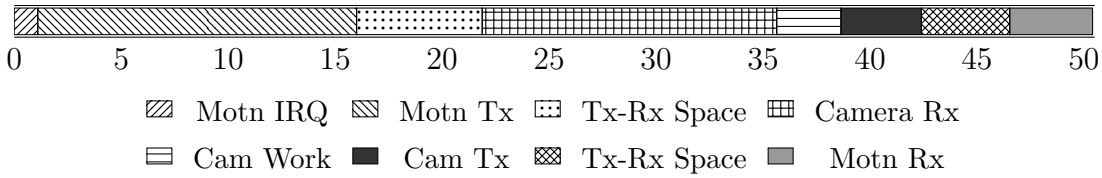

Figure 5.3: Timeline of the events captured using a logic analyzer when the Detection service is deployed on the Edge node and the Motion service is deployed on the Motion node. Units are in milliseconds.

physical hardware, performing neural network inference, storing records in a database, and communicating over 2 incompatible networks requires 402 lines of service code in C++. It also requires 53 lines of Ambience configuration expressed as manifests.

We also test a second deployment in which multiple motion sensors guard the entrances to the camera stage so that image sequences are correct when animals approach from multiple directions. Switching from the 1 motion detector per camera to multiple motion detectors requires 5 lines of configuration change, and no change to the services themselves. Ambience synthesizes the necessary XBee networking code with no additional input from the user.

We deploy the **Detection** service on the camera microcontrollers by default. However if a camera is particularly active causing the battery to drain at a rate that could threaten a shut down, the service can be transparently offloaded to the edge (reducing camera battery load and allowing it to re charge) and then and moved back on-board once the battery is recharged to then reduce load on the edge machine. Performing this reconfiguration manually requires 2 configuration lines to change in Ambience, and no change to the service code itself. While this application is not latency sensitive, latencies on the battery powered nodes have a direct impact on battery life.

5.8.2 Microcontroller Latency Analysis

Overall, microcontrollers' work starting with the motion sensor interrupt to the completion of neural network processing, takes between 19 to 921 milliseconds, (including network transfer) depending on where the Detection and the Motion services are deployed. We show the breakdown of the spent on the microcontroller nodes in Figure 5.3. Note that when the Motion service is deployed on the Camera node, the radio events in the figure do not take place. Neural network inference takes up more than 95% of the time the camera node is awake and we remove it from the trace to make the rest legible. Offloading it to Edge reduces the power use of the node significantly, but increases the load on Edge.

In Figure 5.3, 36 of the 50 milliseconds are devoted to serial communication with radios on each microcontroller board. However, during this time, the microcontroller processors are in low-power sleep and the transfers are made entirely using DMA hardware to conserve power. In particular, the Motion node's microcontroller processor is awake for less than 3% of the entire operation and spends less than 20 μJ for the entire event with the radio spending 4.32 mJ. Using a battery cell with 13Wh capacity [4], this energy expenditure corresponds to approximately 10 million events detected and transmitted over an XBee network. At reasonable event rates and quiescent current leakages, the battery could last multiple years, demonstrating Ambience's abstractions are efficient and effective enough to support low power applications as well as high performance ones.

5.8.3 Edge Platform Overheads

To facilitate reuse and scaling, individual microservices often implement very narrow functionalities, which are composed to form higher level services. Such services are deployed in separate trust domains (processes, address spaces) to achieve isolation.

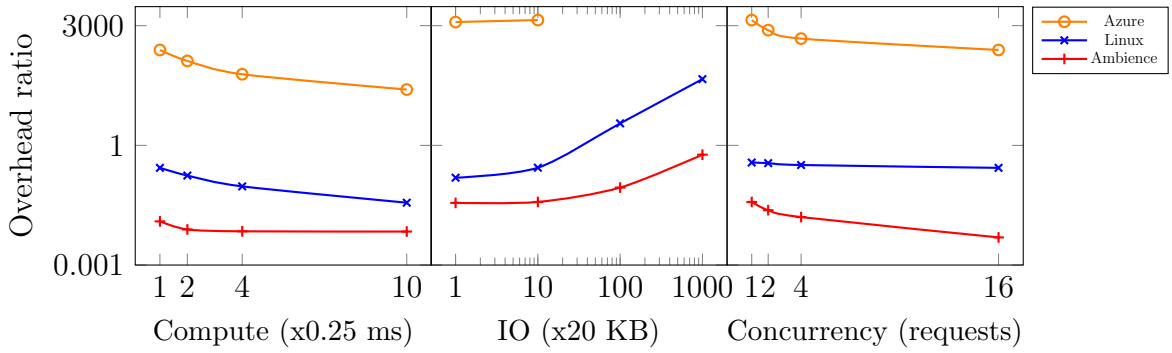

Figure 5.4: Exposition of communication overheads imposed by each platform. Overhead is represented as the ratio of non-compute cycles to compute cycles. Azure has a hard 256KB limit, preventing larger message sizes for it in the IO graph. For the Compute experiment, Concurrency is fixed at 16 and IO is fixed at 200KB. For IO, concurrency is fixed at 1 and work is fixed at 0.25ms. For Concurrency, IO is fixed at 200KB and Compute is fixed at 0.25ms.

Services then use the IPC mechanisms implemented by the operating system to communicate between domains. Previous work [71, 102] notes that microservices can have quite large communication:computation ratios. From a performance perspective, cycles spent for computation is "useful work" and cycles spent for communication is "overhead" – a cost required to implement the useful work. Thus the ratio of communication cycles (cost) to computation cycles (benefit) is a simple representation of the cost/benefit ratio associated with a microservice deployment. We term this metric the *Overhead ratio*.

To study the overhead imposed by the platforms, we implement a pair of Camera and Detection services 3 ways: natively on Ambience, and using Azure IoT SDK's abstractions as well as using Lidl on Linux over Unix Domain Sockets. To make sure all platforms execute the same computation cycle-accurately, we provide parameterizable "mock" versions of both services that allow the workloads to be set explicitly. We then compare the same services across 3 platforms in different configurations. Our mock Detection service allows us to specify an exact cycle count to spend accessing random locations in the given frame. We also experiment with multiple concurrent requests to try and amortize communication costs. Figure 5.4 presents our results.

Our evaluation shows that for low compute, high communication scenarios, Linux imposes significant overheads, spending as high as 61 times the processor cycles for communication than for computation. Azure imposes even higher overheads, spending as much as 1381 cycles on communication per cycle of computation. In comparison, Ambience's overhead ratio ranges from 0.008 to 0.59.

Part of the reason is that typical IPC mechanisms require copying to pass data back and forth between processes. While it works fine for passing small amounts of data between trust domains, it also creates significant overheads relative to the compute happening at individual services. For instance, the **Detection** service requires the entire image to be passed between address spaces, which then executes quickly.

We note from the DeathStar benchmark suite [71] that sub-millisecond computation times per request are common. Also of note is that all systems show improvements as concurrency increases, as communication costs such as i-cache and TLB misses and context switches amortize between concurrent requests, with Ambience improving the most while being the best overall. We expand our analysis of IPC overheads below using microbenchmarks. Note that Azure imposes a hard limit of 256KB [26] on message size, preventing us from probing it for large messages. This experiment demonstrates that Ambience's type-aware, specialized IPC mechanism can dramatically reduce cross-domain overheads for microservices.

5.8.4 Portability of Cloud IoT SDKs

We implemented as much of this system as possible using Azure's SDK as well, as it, like Ambience, features the ability to run on the cloud, edge and microcontrollers. The embedded SDK for microcontrollers is not the same SDK as for the cloud or edge devices, meaning that moving software written for one to the other requires substantive

code changes, testing to make sure that both implementations are equivalent, etc. This impediment is primarily due to divergent APIs, programming and deployment models for the different tiers in a deployment. On the cloud and edge, programmers must make use of Linux APIs, whereas on the microcontroller side, the developer has the choice of FreeRTOS, AzureRTOS or bare metal, each of which provides a unique and incompatible set of abstractions, making writing a piece of code that runs across the platform labor-intensive and error prone.

Even with the equivalent service implementations for different configurations, the communication infrastructure required by the Azure SDK makes use of MQTT for all interservice communication. Thus co-locating two services on the same node incurs MQTT communication overhead even for the local communication. The hard limit on message sizes also makes certain applications impractical on the platform.

The available vendor software did not include support for Xbee radio communication. To the best of the authors' knowledge, the Azure SDK requires the use of IP networks for communication. Thus, to implement the communication between microcontrollers, we had to write a custom XBee driver for the SDK based on the one that is part of Ambience.

Finally, the Azure SDK supports 2 microcontrollers models in the same "family" that we used in the Ambience deployment. However, at the time of the comparative implementation and writing, both of the specific microcontrollers models supported by the vendor were unavailable due to supply-chain delays. We expected the vendor code to be portable to our microcontroller model as well since they share a family but it was not. Thus, qualitatively, since the vendor setup is not portable across 2 microcontrollers that share the same processor core and peripherals, we anticipate that it will be even less portable between platforms. For this reason, we were unable to compare the execution performance of the application end-to-end. We have a comparative implementation from

which we draw the productivity comparisons but the lack of portability prevents a fuller comparison.

5.8.5 Microbenchmarks

IPC Microbenchmark – evaluating static, strongly typed interfaces: In this microbenchmark, we evaluate our integration of the interface type information into the kernel. We created a variety of interfaces to cover a representative set of results. Specifically, we constructed workloads with scalars of a uniform type, scalars of mixed types and relatively large strings and buffers. We executed the benchmarks while increasing the sizes of the arguments to identify any potentially hidden overheads. For each interface, we executed 10K requests using 3 strategies and measured the average latency and overall throughput.

The **User** strategy represents typical interprocess communication (IPC) using bytewise copy between user spaces and the kernel (e.g. Linux pipes). All type serialization occurs in the user space, the kernel copies the bytes from the client to the server and the server deserializes the buffer. For Linux, we implemented the User strategy using pipes on **Linux** 5.15.6 using the same serialization/deserialization code in each comparative experiment.

In the **Dynamic** strategy, the user space sets up a vector of pointers to arguments and tells the kernel the types of the pointers dynamically. The kernel then performs the sharing to the other address space, and creates a new vector of pointers to arguments the server address space can access. The advantage of this approach over the User strategy is the kernel can automatically map pages for large buffers.

The **Static** strategy (the Ambience default strategy) is one in which the user space sets up a tuple of typed arguments and passes a pointer to this tuple to the kernel. Since

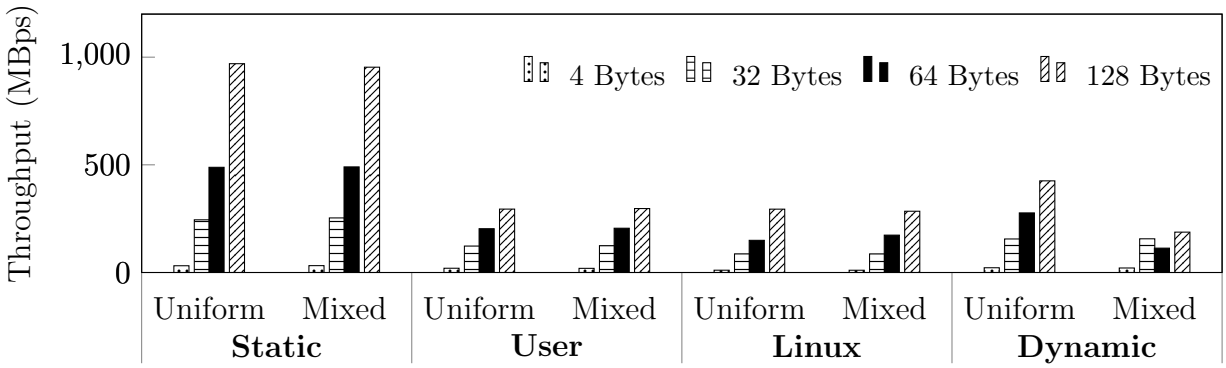

Figure 5.5: Average throughput for passing varying number of scalars between two address spaces. Each experiment is repeated 10,000 times. Static, User and Linux strategy achieve similar results regardless of the argument types, while the Dynamic strategy loses a substantial amount of throughput for mixed types.

the kernel has been compiled with the types information from the interface for the system call it "knows" the structure of the data the tuple. It again creates the same structure on the server address space by either copying the arguments or mapping pages.

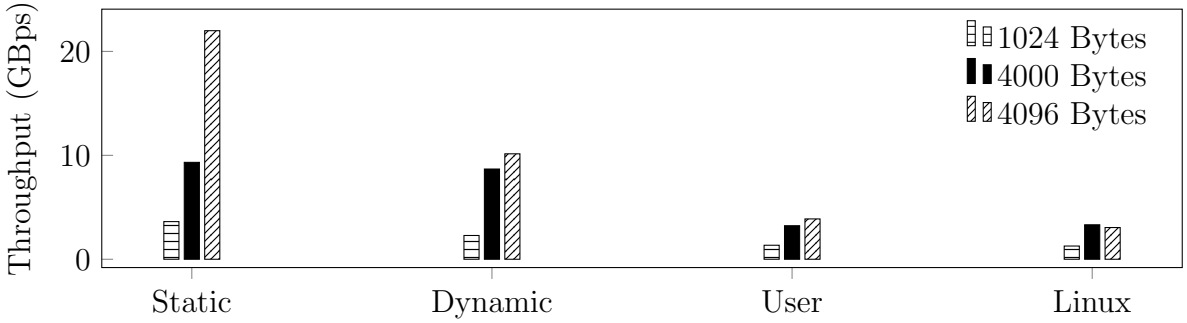

Figure 5.6: Average throughput for passing larger buffers across address spaces. User and Linux fall short as they always perform copies.

We benchmarked our 4 strategies with 11 size classes across 3 clusters:

1. A number of uniform type scalar arguments. For instance `(i64, i64, i64, i64)` for a 32 byte scalar parameter,

2. A number of mixed type scalar arguments. For instance `(u64, i64, bool, i8, f32, i16, f64)` for a 32 byte mixed scalar parameter,

3. A larger byte buffer of sizes 1024, 4000 and 4096 bytes. As the 1024 and 4000 are not

page aligned sizes, they have to be copied for all cases. However, the 4096 byte buffer can be directly mapped if available.

Figures 5.5 and 5.6 show average throughput for the IPC benchmark for scalars and larger input buffers, respectively. The results show that the Static strategy is superior to both approaches, achieving 2.66x and 3.18x higher average and 4.08x and 2.29x higher maximum throughput than Dynamic and User, respectively. As expected, Linux and User results are almost identical results since they are implemented in a very similar manner.

While Dynamic uses the same primitives as Static in a program, for Dynamic, Ambience must traverse a list and make an indirect function call for each argument. The effects of this implementation is most apparent when there are many parameters of different types (i.e. Mixed workload) causing substantial branch mis-predictions and I-Cache trashing. Static, User, and Linux on the other hand have no virtual function calls and there is no list to traverse: parameters are simply a packed, contiguous tuple. On top of the cache-friendliness, the static type information unlocks inlining opportunities for the compiler. For instance, when we are passing 16 scalar arguments, we emit a single large `memcpy` as opposed to 16 small ones. For passing few, well aligned large buffers (starting at around 100KB), Dynamic achieves similar results when its cache and inlining disadvantages get shadowed by page table manipulation.

Because User and Linux need to perform multiple copies of large buffers (one for in-process serialization, another for process to process), they cannot achieve the high throughput afforded by direct page mapping. However, as they can make use of the static types in user space, they still outperform Dynamic for the mixed-type workload.

Note that the automatic page-mapping support cannot be implemented in Linux as it requires automatic support for mapping arbitrary pages from one address space into another, temporarily, with shared ownership. Using `mmap`, two processes could dynami-

cally share pages, however, the users would have to explicitly allocate memory in those pages for the mapping to work, as `mmap` cannot map existing, anonymous pages to another address space. Further, if the same page is supplied as an argument in multiple concurrent requests, the page must be unmapped only when the last request completes. Again, `mmap` does not support mapping the same page multiple times and perform unmap using reference counting.

Scalability Microbenchmark: In this microbenchmark, we evaluate our use of asynchronous programming and coroutines as the primary model. We consider another service in this experiment, a recursive and caching, DNS-like, name resolving service where clients make requests to resolve names to network addresses. We implement the service using both coroutines and fibers to evaluate Ambience's use of stackless coroutines. Specifically, we are interested in their scaling characteristics including the load on memory and compute overheads, if any. To enable this, we deploy 2 terminal resolvers, each storing half of the known domains. We implement one resolver using fibers and the other using coroutines. We also implement one client for each. The clients then generate 10,000 requests for uniformly distributed domains, some of which are invalid.

For the fiber version, we use a stack size of 32KB which is moderate compared to the many megabytes reserved by other systems. The coroutine version does not require such a parameter as each coroutine frame is dynamically allocated.

Because the resolver is recursive, if the requested hostname is not cached, it will make a request to one or more of its upstream resolvers and wait. If the result is in the cache, it responds immediately. Once a request completes, all resources are freed. This means that if a request completes without any blocking, it consumes memory for only a very short time. Therefore, if the cache hit rate is N, only $B * (1 - N)$ requests consume memory in a batch size of B. We aim for $N = 0.5$ in our loads for a realistic workload.

The first graph of Figure 5.7 shows the average throughput for each resolver imple-

mentation for increasing concurrency levels. The results from this and the next graph show that while coroutines improve the memory scalability substantially, they also almost double the throughput for this workload. However, fibers and coroutines have dramatically different memory allocation patterns which makes their performance characteristics highly workload dependent.

The second graph of Figure 5.7 shows the average memory use in bytes for each resolver implementation for increasing concurrency levels. Fibers have a constant per-request memory cost of 32KB as expected, while coroutines need only allocate the bare minimum memory associated with a given request, which in this experiment is 627 bytes. With fibers, it is not possible to precisely allocate enough memory with any recursion/indirection. With the same amount of memory, coroutines can maintain more than 50x requests.

Note that the top graph of Figure 5.7 shows an increase in throughput up to a concurrency of 64 for coroutines, after which the throughput starts to decline. To investigate this phenomenon in detail, we implemented kernel support for the Performance Monitor Counters for the 5950x processor and gathered information on cache and TLB misses.

For fewer than 64 coroutines, the increase in throughput stems from a dramatic reduction in the number of context switches and TLB and cache misses as shown in the third graph of Figure 5.7. However, every additional concurrent request grows the working set as individual, dedicated pages are created for each request and response, and after a point, the increasing cache misses causes throughput to decline as shown in the last graph of 5.7.

Group Microbenchmark: Using its group abstraction, Ambience is able to place multiple logically isolated components in a single address space. To evaluate this design choice, we placed the coroutine recursive resolver in the same group as the client. We keep the backend resolvers in different address spaces as before. We show the throughput

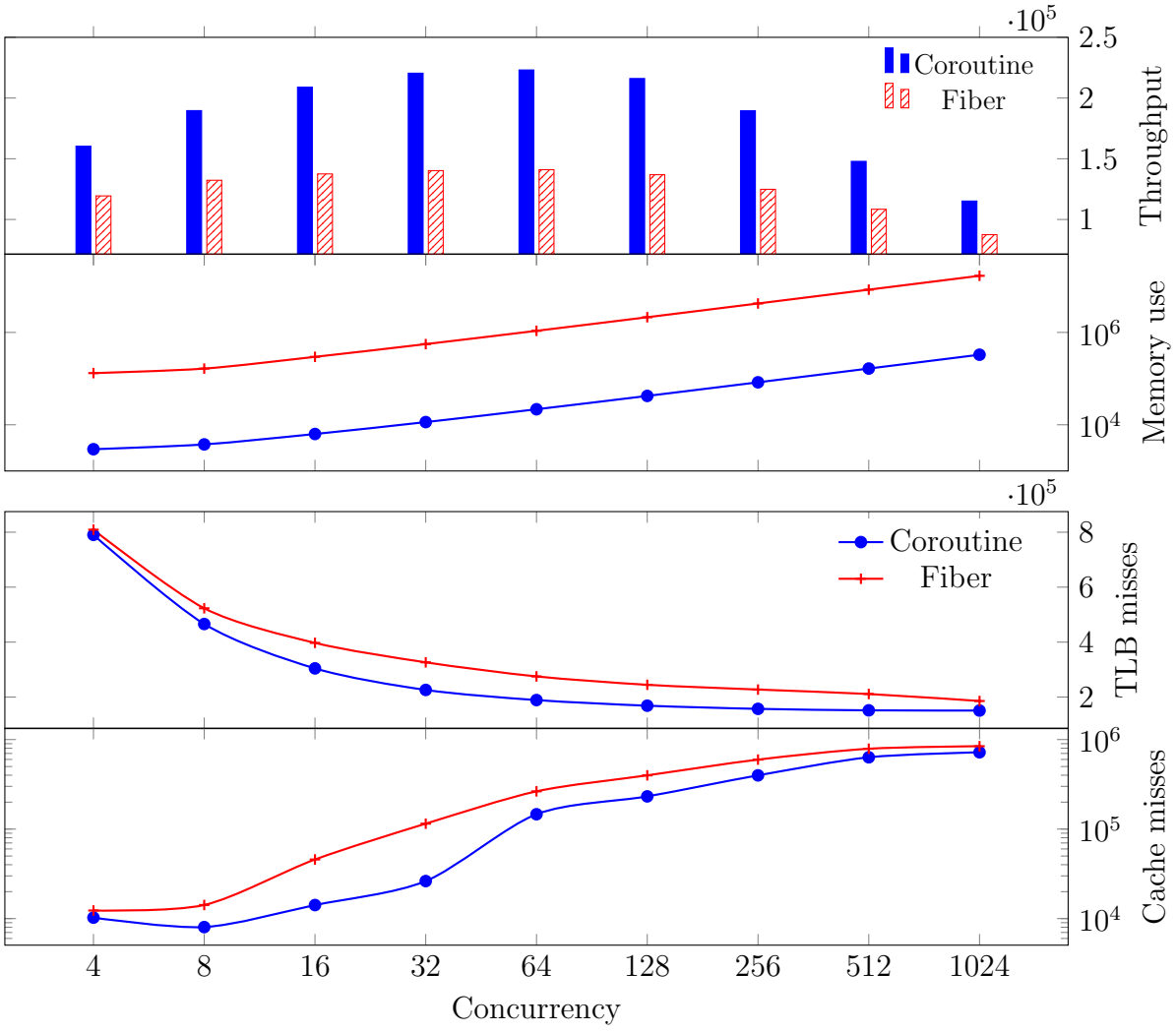

Figure 5.7: Request throughput (QPS), memory use (bytes), TLB and Cache misses as concurrency increases

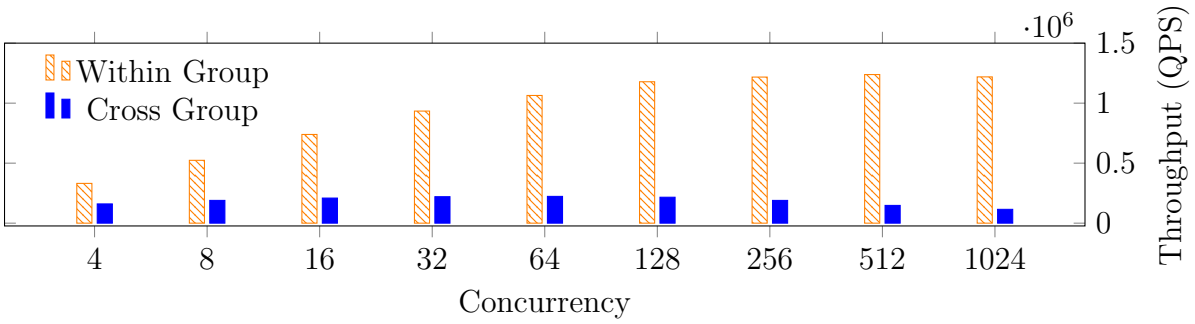

Figure 5.8: Request throughput as concurrency increases, when the resolver and the client are deployed in the same (Within) and different (Cross) groups.

results of this experiment in Figure 5.8. Without any modification to either the resolver or client, Ambience achieves up to an order of magnitude higher average throughput when the services are placed in the same group and are optimized by Ambience.

Scheduling Microbenchmark – time to first instruction latency: Ambience allows for extremely flexible deployment strategies, one of which is the opportunity to place an unmodified service in either user space or directly in kernel. As context switch costs grow, such an option becomes highly desirable for low-latency applications. In this microbenchmark, we evaluate the benefits of this flexibility by measuring the time from an interrupt handler that triggers a service to the execution of the first instruction of the service.

When the service is in-kernel, Ambience can immediately schedule the service on the kernel job queue, whereas when it is in user space, it must set up various protection structures and enqueue the service for scheduling. Once scheduled after a context switch, the service is finally jumped to.

We present the results (average microseconds) in Table 5.2. This Ambience optimization reduces startup latency by 27x for the edge system and by 8x for the microcontroller.

	User Space	Kernel
Edge PC	9.418	0.348
Motion MCU	58.765	7.265

Table 5.2: Time to first service instruction from a hardware interrupt on Edge and Motion nodes when the service is deployed in user space or in kernel. Units are microseconds.

5.9 Summary

We present Ambience, an operating system for efficiently executing and deploying microservices. It does so via a novel combination of abstractions for isolation with locality control, asynchronous user space, statically typed interfaces, automatic networking and access control and declarative deployment orchestration. This combination makes it possible to optimize not only individual services but the kernels running those services and reduce the overheads that hamper the use of general purpose operating systems on resource constrained machines.

Ambience's streamlined design and focus on footprint size and low resource use make it possible to execute microservices portably across a wide range of resource types and scales, e.g. microcontrollers, single board computers, edge systems, cloud computing instances, and Linux systems.

Our end to end experiment demonstrates microservices' viability to provide a unified programming model for IoT and other heterogeneous hardware applications efficiently. Our microbenchmarks show that Ambience's divergence from the mainstream in user space design and use of static information can provide significant performance gains. Moreover, our results show that Ambience is able to scale down to very resource restricted devices while scaling to cloud systems. Finally, Ambience is able to do so using identical operating system abstractions across all tiers.

Ambience's highly optimizable design and efficient implementation fully realizes our Devices-as-services vision first presented in the introduction of this thesis, where the

entirety of a distributed Internet of Things application, spanning vastly heterogeneous hardware configurations and diverse networks can be built using the same set of abstractions.

Chapter 6

Conclusion and Future Work

In this thesis, we present Ambience, an operating system for efficiently executing and deploying microservices. To meet the efficiency requirements necessitated by the portability requirement, Ambience diverges from mainstream operating system designs in key areas. First, Ambience has a novel user space design tailored specifically for hosting event driven microservices. Ambience's user space - kernel space interfaces are strongly typed, allowing Ambience to manage inter service communication automatically. Second, Ambience decouples isolation from deployment, allowing multiple, not-necessarily-cooperative services to be hosted within the same address space to recover performance at deployment time. Third, the user space type information is also used to drive optimizations of individual kernel images. Ambience automatically synthesizes optimal interservice communication code to be embedded inside kernels. The optimal interservice communication achieves throughputs multiple orders of magnitude higher than the equivalent Linux version of the same service.

Ambience services are secured using CAPLets, an efficient authorization framework designed for resource constrained and rich devices. CAPLets uses an efficient cryptographic construction based on Message Authentication Codes to enable execution on

microcontrollers. Policies for CAPLets can be expressed in turing complete languages, allowing the implementation of arbitrary, dynamic access control policies. Combined with the decentralized nature of its design, CAPLets enables truly flexible and decentralized authorization for the Internet of Things.

Ambience supports and provides the same programming environment and abstractions on microcontrollers and cloud servers alike, implementing the Devices as Services model.

Work described in this thesis makes the following, non-exhaustive list of notable contributions to their respective fields:

State of the art IoT programming frameworks and platforms [25, 98, 97, 78, 131] treat microcontrollers as second class citizens, leaving performance, reliability and cost improvements on the table. Devices as services, explained in Chapter 3, provides a structured and principled method for programming IoT applications uniformly.

CSPOT, EdgePy and NanoLambda, discussed in in Chapter 3, implement this model practically to allow the execution of C, C++ and Python programs on microcontrollers and cloud servers alike. Previous work [78, 20, 130] cannot bridge the gap between the edge and microcontrollers.

State of the art authorization mechanisms in IoT applications [65, 82] make use of expensive cryptographic primitives based on web technologies. These primitives cannot efficiently scale down to resource-poor devices. CAPLets, detailed in Chapter 4, is designed and implemented with power aware primitives in mind to allow execution on even the weakest microcontrollers.

Previous security primitives couple the arguably orthogonal concerns of security and privacy [154]. CAPLets securely decouples the two to allow application developers to pick the optimal approach given their specific resource and feature requirements.

Existing key exchange protocols always make use of costly public key cryptography

identities to provide authentication of parties. CAPLets introduces a secure key exchange protocol based on CAPLets tokens, only using cheap primitives to enable efficient application on low end microcontrollers.

Elimination of overheads of general purpose operating systems for specific workloads have been a research topic for as long as general purpose operating systems existed. Past work includes exokernels [64], runtime code synthesis for specialization [151, 38] and more recently unikernels [122, 44, 140, 113] and kernel bypass systems [34, 148, 60, 100]. The common theme across this body of work is the elimination of kernel's responsibilities and moving them to user controlled code. Ambience acknowledges the problem, but attempts to eliminate overheads by specializing the kernel for the task at hand, as the kernel space is often the most efficient environment for implementing operating system level abstractions.

Mainstream microservice platforms [112, 58, 131] converged on the use of immutable, declarative manifests for configuring entire clusters. However, all these systems are built on general purpose platforms, which cannot take advantage of the information within these manifests. Ambience unlocks novel optimizations in services and inside the kernel images by exploiting the information in static declarations.

Previous work [38, 94] on use of static type information in operating systems have focused their efforts on making use of program verification using strong types to achieve software fault isolation. Such use invariably prevents the use of arbitrary programming languages, and often require the programmer to program in an unpopular research programming language. Ambience makes novel use of types to drive performance optimizations inside services and the kernels that host the services. Ambience does not trust the interfaces in any way, allowing the implementation of services in any programming language.

In mainstream microservice systems, the unit of deployment is the process (in fact,

it is the container), same as the unit of operating system provided isolation. This means that once a service is decoupled from another, they have to be deployed separately without code changes. This stiff deployment model creates design and efficiency challenges. Ambience decouples isolation from deployment, where programmers can create flexible isolation boundaries at *deployment time* to deploy multiple, non-cooperative services in the same trust domain to recover performance.

Ambience can be extended in many ways. Currently, Ambience is too static. First, while new services can be added to an Ambience node, the new services must have an interface that the running kernel already knows about (i.e. compiled in at deployment time). If not, then the service cannot be deployed at all without a kernel update, which requires a reboot. To make Ambience more practical in real world uses and increase its availability, some dynamism must be added. Second, currently, RPC style request-response services are the only communication primitive, which can be limiting in certain applications. Specifically, Ambience must support pub-sub style applications. While a direct port of, for instance, MQTT to Ambience is possible, it would not fit well within Ambience. A new design, or adaptation of an existing system that embraces Ambience's approach to efficiency and portability could open up new applications on Ambience. Finally, Ambience needs to support more service topologies than static, acyclic dependencies. The current design, for instance, precludes the implemenetation of common distributed consensus algorithms as they require cyclical dependencies to work. With these features designed and implemented, Ambience can support almost every real world microservice and IoT application efficiently across all platforms.

Bibliography

[1] Json web token (jwt).

[2] Transmission control protocol. RFC 793, IETF, September 1981.

[3] Amqp home page. https://www.amqp.org, 2019. [Online; accessed 2-May-2019].

[4] Panasonic 18650 battery datasheet. `https://www.imrbatteries.com/content/panasonic_ncr18650b-2.pdf`.

[5] Leila Abdollahi Vayghan, Mohamed Aymen Saied, Maria Toeroe, and Ferhat Khendek. Deploying microservice based applications with kubernetes: Experiments and lessons learned. In *International Conference on Cloud Computing (CLOUD)*, page 970–973, Jul 2018.

[6] Carlisle Adams and Steve Lloyd. *Understanding public-key infrastructure: concepts, standards, and deployment considerations*. Sams Publishing, 1999.

[7] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight virtualization for serverless applications. In *17th {usenix} symposium on networked systems design and implementation ({nsdi} 20)*, pages 419–434, 2020.

[8] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Eurosys*, page 30, 2018.

[9] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. Understanding the mirai botnet. In *USENIX Security Symposium*, August 2017.

[10] Boost.asio, 2021. `https://www.boost.org/doc/libs/1_76_0/doc/html/boost_asio.html`.

[11] JSON Authors. Json (javascript object notation). `https://www.json.org/json-en.html`.

[12] Linux Kernel Authors. Bpf documentation. https://www.kernel.org/doc/html/latest/bpf/index.html, 2021.

[13] Linux Kernel Authors. Efficient io with io_uring. *Linux Documentation*, page 16, 2021.

[14] Quarkus Authors. Quarkus, a kubernetes native java stack.

[15] Spring Authors. Nestjs.

[16] Spring Authors. Spring boot.

[17] Wasm3 Authors. Wasm3. https://github.com/wasm3/wasm3, 2020.

[18] Abel Avram. FaaS, PaaS, and the Benefits of the Serverless Architecture. https://www.infoq.com/news/2016/06/faas-serverless-architecture. [Online; accessed 15-Nov-2016].

[19] AWS Greengrass. https://aws.amazon.com/greengrass/ [Online; accessed 12-Sep-2019].

[20] AWS Lambda. https://aws.amazon.com/lambda/. [Online; accessed 15-Nov-2016].

[21] AWS Lambda IoT Reference Architecture. http://docs.aws.amazon.com/lambda/latest/dg/lambda-introduction.html [Online; accessed 1-Nov-2016].

[22] Aws lambda – serverless compute - amazon web services. https://aws.amazon.com/lambda/.

[23] axtls. http://axtls.sourceforge.net/ [Online; accessed on 14-June-2022].

[24] Azure Functions. https://azure.microsoft.com/en-us/services/functions/. [Online; accessed 15-Nov-2016].

[25] Azure Internet of Things. https://www.microsoft.com/en-us/cloud-platform/internet-of-things-azure-iot-suite. [Online; accessed 22-Aug-2016].

[26] Understand azure iot hub quotas and throttling — microsoft docs. https://github.com/MicrosoftDocs/azure-docs/blob/4764ddf7bb4c71e33e58dfe67dbd7361ee0fb6fa/includes/iot-hub-limits.md.

[27] Fatih Bakir, Chandra Krintz, and Rich Wolski. Caplets: Resource aware, capability-based access control for iot. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 106–120. IEEE, 2021.

[28] Fatih Bakir, Rich Wolski, Chandra Krintz, and Gowri Sankar Ramachandran. Devices-as-services: Rethinking scalable service architectures for the internet of things. In *2nd USENIX Workshop on Hot Topics in Edge Computing (HotEdge 19)*, Renton, WA, July 2019. USENIX Association.

[29] Fatih Bakir, Rich Wolski, Chandra Krintz, and Gowri Sankar Ramachandran. Devices-as-services: Rethinking scalable service architectures for the internet of things. In *USENIX HotEdge*, 2019.

[30] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. M. Rabbah, A. Slominski, and P. Suter. Serverless computing: Current trends and open problems. *CoRR*, 2017.

[31] A. Banks and R. Gupta. Mqtt v3.1.1 protocol specification, 2014.

[32] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[33] Elaine Barker, William Barker, William Burr, William Polk, and Miles Smid. Recommendation for key management – part 1: General (revision 4). *NIST Special Publication Revision*, 01 2016.

[34] Adam Belay, George Prekas, Christos Kozyrakis, Ana Klimovic, Samuel Grossman, and Edouard Bugnion. Ix: A protected dataplane operating system for high throughput and low latency. In *USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[35] Nicole Berdy. How to use Azure Functions with IoT Hub message routing, 2017. "https://azure.microsoft.com/en-us/blog/how-to-use-azure-functions-with-iot-hub-message-routing/".

[36] Mark Bergen. Google outage reignites worries about smart home without backups.

[37] Daniel J Bernstein. Curve25519: new diffie-hellman speed records. In *International Workshop on Public Key Cryptography*, pages 207–228. Springer, 2006.

[38] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, S. Eggers, and C. Chambers. Extensibility, Safety, and Performance of the SPIN Operating System. In *Symposium on Operating System Principles*, December 1995.

[39] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, and Susan J. Eggers. Extensibility, safety and performance in the SPIN operating system. In *SOSP*, 1995.

[40] R. Bias. The history of pets vs cattle and how to use the analogy properly, Sep 2016. http://cloudscaling.com/blog/cloud-computing/the-history-of-pets-vs-cattle/.

[41] Arnar Birgisson, Joe Gibbs Politz, Ulfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. 2014.

[42] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Network and Distributed System Security Symposium*, 2014.

[43] Carsten Bormann and Paul Hoffman. Concise binary object representation (cbor). Technical report, RFC 7049, DOI 10.17487/RFC7049, October 2013,¡ https://www. rfc-editor. org . . . , 2013.

[44] Alfred Bratterud, Alf-Andre Walla, Harek Haugerud, Paal E. Engelstad, and Kyrre Begnum. Includeos: A minimal, resource efficient unikernel for cloud services. In *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, page 250–257, Nov 2015.

[45] Zach Brown. Asynchronous system calls. In *Proceedings of the Ottawa Linux Symposium (OLS)*, pages 81–85, 2007.

[46] Ethan Buchman. *Tendermint: Byzantine fault tolerance in the age of blockchains.* PhD thesis, University of Guelph, 2016.

[47] The centos project, 2021. https://www.centos.org/.

[48] Cloud application platform — heroku. https://www.heroku.com/.

[49] Component Object Model. "https://docs.microsoft.com/en-us/windows/win32/com/component-object-model–com–portal".

[50] constexpr specifier - cppreference.com, 2021. https://en.cppreference.com/w/cpp/language/constexpr.

[51] Fedora coreos, 2021. https://getfedora.org/en/coreos?stream=stable.

[52] craigshoemaker. Azure functions overview. https://docs.microsoft.com/en-us/azure/azure-functions/functions-overview.

[53] Debian gnu/linux. "http://www.debian.org".

[54] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swami Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's Highly Available Key-Value Store. In *Symposium on Operating System Principles*, 2007.

[55] DevOps. `http://en.wikipedia.org/wiki/DevOps`, 2013. [Online; accessed 27-September-2013].

[56] D. Jeff Dionne and Kenneth Albanowski. μclinux. `https://en.wikipedia.org/wiki/%CE%9CClinux`.

[57] Digitalocean. `https://digitalocean.com/`.

[58] Docker compose. `https://docs.docker.com/compose/` [Online; accessed on 14-June-2022].

[59] Docker compose, Aug 2021.

[60] Dpdk home. `https://www.dpdk.org/`.

[61] Elliptic-curve cryptography. `https://en.wikipedia.org/wiki/Elliptic-curve_cryptography`.

[62] EEMBC. Coremark, an eembc benchmark. `https://www.eembc.org/coremark/`, 2021.

[63] Andy Rosales Elias, Nevena Golubovic, Chandra Krintz, and Rich Wolski. Wheres the bear?–automating wildlife image processing using iot and edge cloud systems. In *ACM Conference on IoT Design and Implementation*, 2017.

[64] D. Engler, M. Kaashoek, and J. O'Toole Jr. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Symposium on Operating System Principles*, December 1995.

[65] Andres Erbsen, Asim Shankar, and Ankur Taly. Distributed authorization in vanadium. *arXiv preprint arXiv:1607.02192*, 2016.

[66] ESP8266 Specifications Web Site, 2019. [Online; accessed 15-March-2019] `https://www.espressif.com/en/products/hardware/esp8266ex/overview`.

[67] V. Eswara, G. Srivastava, and S. Biswas. Riotnet: Reactive iot control network. In *IEEE International Conference on Internet of Things*, June 2017.

[68] Fuchsia Interface Definition Language. "https://fuchsia.dev/fuchsia-src/development/languages/fidl".

[69] Fog Data Services - Cisco. http://www.cisco.com/c/en/us/products/cloud-systems-management/fog-data-services/index.html. [Online; accessed 22-Aug-2016].

[70] Freertos - market leading rtos, 2021. https://www.freertos.org/index.html.

[71] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, et al. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 3–18, 2019.

[72] Wolfgang Gentzsch, Lucio Grandinetti, and Gerhard Robert Joubert. *High speed and large scale scientific computing*, volume 18. IOS Press, 2009.

[73] Gareth George, Fatih Bakir, Rich Wolski, and Chandra Krintz. Nanolambda: Implementing functions as a service at all resource scales for the internet of things. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*, page 220–231, Nov 2020.

[74] Google. Google psp. https://github.com/google/psp/.

[75] Google. Flatbuffers. https://google.github.io/flatbuffers/, 2021.

[76] Google Cloud Functions. https://cloud.google.com/functions/docs/. [Online; accessed 15-Nov-2016].

[77] Cloud functions, 2021. https://cloud.google.com/functions.

[78] GreenGrass and IoT Core - Amazon Web Services. https://aws.amazon.com/iot-core,greengrass/. [Online; accessed 2-Mar-2019].

[79] gRPC, 2021. https://grpc.io/ [Online; accessed 1-Aug-2021].

[80] S. Gusmeroli, S. Piccione, and D. Rotondi. Iot access control issues: A capability based approach. In *2012 Sixth International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing*, pages 787–792, 2012.

[81] S. Gusmeroli, S. Piccione, and D. Rotondi. A Capability-based Security Approach to Manage Access Control in the Internet of Things. *Mathematical and Computer Modelling*, 58(5-6), 2013.

[82] Sergio Gusmeroli, Salvatore Piccione, and Domenico Rotondi. A capability-based security approach to manage access control in the internet of things. *Mathematical and Computer Modelling*, 58(58):1189–1205, 2013.

[83] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and Jf Bastien. Bringing the web up to speed with webassembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2017*, page 185–200. ACM Press, 2017.

[84] Taylor Hardin, Josiah Hester, Ryan Scott, Jacob Sorber, Patrick Proctor, and David Kotz. Application memory isolation on ultra-low-power mcus. In *USENIX Annual Technical Conference*, 2018.

[85] Hc-sr501 pir motion detector. `https://www.mpja.com/download/31227sc.pdf`.

[86] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. In *Conference on Innovative Data Systems Research (CIDR)*, 2019.

[87] Helm. `https://helm.sh/`.

[88] S. Hendrickson, S. Sturdevant, T. Harter, V. Venkataramani, A. Arpaci-Dusseau, and R. Arpaci-Dusseau. Serverless computation with openlambda. In *HotCloud*, 2016.

[89] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Serverless computation with openlambda. In *HotCloud*, 2016.

[90] J Hernandez-Ramos, A. Jara, L. Marin, and A. Skarmeta Gomez. Dcapbac: Embedding authorization logic into smart things through ecc optimizations. *Int. J. Comput. Math.*, 93(2), February 2016.

[91] J. L. Hernandez-Ramos, A. J. Jara, L. Marin, and A. F. Skarmeta Gomez. Dcapbac: embedding authorization logic into smart things through ecc optimizations. *International Journal of Computer Mathematics*, 93(2):345–366, 2016.

[92] José L Hernández-Ramos, Antonio J Jara, Leandro Marin, and Antonio F Skarmeta. Distributed capability-based access control for the internet of things. *Journal of Internet Services and Information Security (JISIS)*, 3(3/4):1–16, 2013.

[93] Dan Hildebrand. An architectural overview of qnx. In *USENIX Workshop on Microkernels and Other Kernel Architectures*, pages 113–126, 1992.

[94] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *Operating Systems Review*, 41(2):37–49, 2007.

[95] Galen C Hunt and James R Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

[96] Intel NUC. https://en.wikipedia.org/wiki/Next_Unit_of_Computing [Online; accessed 1-Feb-2018].

[97] Internet of Things - Amazon Web Services. https://aws.amazon.com/iot/. [Online; accessed 22-Aug-2016].

[98] Internet of Things Solutions - Google Cloud Platform. https://cloud.google.com/solutions/iot/. [Online; accessed 22-Aug-2016].

[99] iRobot. An amazon aws outage is currently impacting our irobot home app...

[100] EunYoung Jeong, Shinae Woo, Muhammad Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. mtcp: A highly scalable user-level tcp stack for multicore systems. In *USENIX Symposium on Networked Systems Design and Implementation*, 2014.

[101] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 152–166, 2021.

[102] Zhipeng Jia and Emmett Witchel. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, page 152–166. ACM, Apr 2021.

[103] E. Jonas, J. Schleier-Smith, V. Sreekanti, C. Tsai, A. Khandelwal, Q. Pu, V. Shankar, J. Carreira, K. Krauth, N. Yadwadkar, J. Gonzalez, R. Popa, I. Stoica, and D. Patterson. Cloud Programming Simplified: A Berkeley View on Serverless Computing, Feb 2019.

[104] Christopher Jonathan, Umar Farooq Minhas, James Hunter, Justin Levandoski, and Gor Nishanov. Exploiting coroutines to attack the" killer nanoseconds". *Proceedings of the VLDB Endowment*, 11(11):1702–1714, 2018.

[105] J. Jonsson and B. Kaliski. Rfc3447: Public-key cryptography standards (pkcs) #1: Rsa cryptography specifications version 2.1. Technical report, Network Working Group, The Internet Society, 2003.

[106] M. Jung, S. Mollering, P. Dalbhanjan, P. Chapman, and C. Kassan. Microservices on AWS. https://docs.aws.amazon.com/aws-technical-content/latest/microservices-on-aws/introduction.html, September 2017. [Online; accessed 2-Mar-2019].

[107] Karl-Bridge-Microsoft. Fibers - win32 apps. https://docs.microsoft.com/en-us/windows/win32/procthread/fibers.

[108] Cameron F Kerry and Charles Romine Director. Fips pub 186-4 federal information processing standards publication digital signature standard (dss). 2013.

[109] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[110] Ronny Ko and James Mickens. Deadbolt: Securing iot deployments. In *Proceedings of the Applied Networking Research Workshop on*, pages 50–57, 2018.

[111] Hugo Krawczyk, Ran Canetti, and Mihir Bellare. Hmac: Keyed-hashing for message authentication, 1997. [Online; accessed 26-Apr-2019] `https://tools.ietf.org/html/rfc2104`.

[112] Kubernetes. `https://kubernetes.io/`.

[113] Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Ştefan Teodorescu, Costi Răducanu, et al. Unikraft: fast, specialized unikernels the easy way. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 376–394, 2021.

[114] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis and transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–88, San Jose, CA, USA, Mar 2004.

[115] H. Lee, K. Satyam, and G. Fox. Evaluation of production serverless computing environments. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, July 2018.

[116] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kb computer safely and efficiently. In *Symposium on Operating Systems Principles*, 2017.

[117] libevent, 2021. `https://libevent.org/`.

[118] Welcome to the libuv documentation — libuv documentation, 2021. `http://docs.libuv.org/en/v1.x/`.

[119] Z. Ling, J. Luo, Y. Xu, C. Gao, K. Wu, and X. Fu. Security vulnerabilities of internet of things: A case study of the smart plug system. *IEEE Internet of Things Journal*, 4(6), Dec 2017.

[120] L. Liu. Privacy and location anonymization in location-based services. *SIGSPATIAL Special*, 1(2), July 2009.

[121] T. Lynn, P. Rosati, A. Lejeune, and V. Emeakaroha. A preliminary review of enterprise serverless cloud computing (function-as-a-service) platforms. In *IEEE International Conference on Cloud Computing*, Dec 2017.

[122] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *ACM SIGARCH Computer Architecture News*, 41(1), 2013.

[123] Parikshit N. Mahalle, Bayu Anggorojati, Neeli R. Prasad, and Ramjee Prasad. Identity authentication and capability based access control (iacac) for the internet of things. *Journal of Cyber Security and Mobility*, 1(4):309–348, 2012.

[124] Parikshit N Mahalle, Bayu Anggorojati, Neeli R Prasad, Ramjee Prasad, et al. Identity authentication and capability based access control (iacac) for the internet of things. *Journal of Cyber Security and Mobility*, 1(4):309–348, 2013.

[125] Mbed-tls. `https://github.com/Mbed-TLS/mbedtls` [Online; accessed on 14-June-2022].

[126] G. McGrath and P. R. Brenner. Serverless computing: Design, implementation, and performance. In *International Conference on Distributed Computing Systems Workshops*, June 2017.

[127] Maged Michael, Jose E Moreira, Doron Shiloach, and Robert W Wisniewski. Scale-up x scale-out: A case study using nutch/lucene. In *2007 IEEE International Parallel and Distributed Processing Symposium*, pages 1–8. IEEE, 2007.

[128] ST Microelectronics. Stm32f7 - very high-performance mcus with cortex-m7. `https://www.st.com/en/microcontrollers-microprocessors/stm32f7-series.html`.

[129] ST Microelectronics. Stm32h7 - arm cortex-m7 and cortex-m4 mcus (480 mhz). `https://www.st.com/en/microcontrollers-microprocessors/stm32h7-series.html`.

[130] Micropython. `https://microPython.org` [Online; accessed on 24-June-2020].

[131] Microsoft. Iot edge: Microsoft azure.

[132] Microsoft. Iot hub: Microsoft azure.

[133] Microsoft Interface Definition Language. "https://docs.microsoft.com/en-us/windows/win32/midl/midl-start-page".

[134] Iot edge — microsoft azure. `https://azure.microsoft.com/en-us/services/iot-edge/`.

[135] S. Mullender, G. van Rossum, A. Tanenbaum, R. van Renesse, and H. van Staveren. Amoeba – A distributed Operating System for the 1990's. *IEEE Computer*, 23(5), May 1990.

[136] Irakli Nadareishvili, Ronnie Mitra, Matt McLarty, and Mike Amundsen. *Microservice architecture: aligning principles, practices, and culture.* " O'Reilly Media, Inc.", 2016.

[137] Node.js. Node.js, 2021. `https://nodejs.org/en/`.

[138] M. Noura, M. Atiquzzaman, and M. Gaedke. Interoperability in internet of things: Taxonomies and open challenges. *Mobile Network Applications*, 24, 2019.

[139] Daniel Nurmi, Richard Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. The eucalyptus open-source cloud-computing system. In *IEEE Cluster Computing and the Grid*, 2009.

[140] Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min, and Binoy Ravindran. A binary-compatible unikernel. In *ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, page 59–73, 2019.

[141] Openfaas. `https://docs.openfaas.com/` [Online; accessed on 14-June-2022].

[142] OpenLDAP. "http://www.openldap.org/".

[143] Aafaf Ouaddah, Anas Abou Elkalam, and Abdellah Ait Ouahman. Fairaccess: a new blockchain-based access control framework for the internet of things. *Security and communication networks*, 9(18):5943–5964, 2016.

[144] Ov5640 datasheet. `https://cdn.sparkfun.com/datasheets/Sensors/LightImaging/OV5640_datasheet.pdf`.

[145] B. Pearson, L. Luo, Y. Zhang, R. Dey, Z. Ling, M. Bassiouni, and X. Fu. On misconception of hardware and cost in iot security and privacy. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, pages 1–7, May 2019.

[146] Boris Pismenny, Ilya Lesokhin, Liran Liss, and Haggai Eran. Tls offload to network devices. In *The Technical Conference on Linux Networking (Netdev)*, 2016.

[147] Thomas Pornin. Bearssl.

[148] George Prekas, Marios Kogias, and Edouard Bugnion. Zygos: Achieving low tail latency for microsecond-scale networked tasks. In *Proceedings of the 26th Symposium on Operating Systems Principles*, page 325–341. ACM, Oct 2017.

[149] Protocol Buffers. Google's Data Interchange Format. "http://code.google.com/p/protobuf".

[150] Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[151] Calton Pu, Henry Massalin, and John Ioannidis. The synthesis kernel. *Computing Systems*, 1(1):11–32, 1988.

[152] Rancher os, 2021. `https://rancher.com/`.

[153] Representational State Transfer (REST). `https://en.wikipedia.org/wiki/Representational_state_transfer` [Online; accessed 1-Nov-2016].

[154] E. Rescorla. The transport layer security (tls) protocol version 1.3. RFC 8446, IETF, August 2018.

[155] Dennis M Ritchie and Ken Thompson. The unix time-sharing system. *Bell System Technical Journal*, 57(6):1905–1929, 1978.

[156] Raspberry pi. `https://www.raspberrypi.com/` [Online; accessed on 14-June-2022].

[157] Rusty Russell. virtio: towards a de-facto standard for virtual i/o devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, Jul 2008.

[158] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of The IEEE*, 63(9):1278–1308, 1975.

[159] Jerome H. Saltzer. Protection and the control of information sharing in multics. *Communications of The ACM*, 17(7):388–402, 1974.

[160] Sandstorm.io. Cap'n proto. `https://capnproto.org/`, 2021.

[161] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4), 2009.

[162] Serverless Platform. [Online; accessed 10-Feb-2019] `www.serverless.com`.

[163] Amazon Web Services. Aws fargate, serverless compute for containers.

[164] X. Shelby, K. Hartke, and C. Borman. The Constrained Application Protocol (CoAP). RFC 7252, IETF, 2014.

[165] M. Slee, A. Aarwal, and M. Kwiatkowski. Thrift: Scalable Cross-Language Services Implementation (White Paper), April 2007. `http://incubator.apache.org/thrift/`.

[166] Livio Soares. Flexsc: Flexible system call scheduling with exception-less system calls. In *USENIX Symposium on Operating Systems Design and Implementation*, 2010.

[167] Livio Soares. Exception-less system calls for event-driven servers. In *USENIX Annual Technical Conference*, 2011.

[168] Stack overflow developer survey 2021.

[169] A. Stanford-Clark and H. Truong. Mqtt for sensor networks (mqtt-sn) protocol specification, 2013.

[170] Inc. Sun Microsystems. White paper: Java(TM) 2 Platform Micro Edition(J2ME(TM)) Technology for Creating Mobile Devices, May 2000. `http://java.sun.com/products/cldc/wp/ KVMwp.pdf`.

[171] Terraform by hashicorp. `https://www.terraform.io/`.

[172] Tensorflow lite — ml for mobile and edge devices. `https://www.tensorflow.org/lite/`.

[173] Rahmadi Trimananda, Ali Younis, Bojun Wang, Bin Xu, Brian Demsky, and Guoqing Xu. Vigilia: Securing smart home edge computing. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 74–89, 2018.

[174] uBPF Authors. ubpf. `https://github.com/iovisor/ubpf`, 2021.

[175] Ubuntu server - for scale out workloads, 2021. `https://ubuntu.com/server`.

[176] Tim Verbelen, Pieter Simoens, Filip De Turck, and Bart Dhoedt. Cloudlets: Bringing the cloud to the mobile user. In *Proceedings of the third ACM workshop on Mobile cloud computing and services*, pages 29–36, 2012.

[177] Virtualbox. `https://www.virtualbox.org/`.

[178] Dr. Thorsten von Eicken's Low Power WiFi Blog, 2019. [Online; accessed 15-March-2019] `https://blog.voneicken.com/projects/low-power-wifi-intro/`.

[179] WebAssembly Community Group. *WebAssembly Specification*, 2020. Version 1.

[180] Windows server 2019 — microsoft, 2021. `https://www.microsoft.com/en-us/windows-server`.

[181] R. Wolski and C. Krintz. CSPOT: A Serverless Platform of Things. Technical Report 2018-01, UC Santa Barbara, 2018. `https://www.cs.ucsb.edu/research/tech-reports/2018-01`.

[182] Rich Wolski, Chandra Krintz, Fatih Bakir, Gareth George, and Wei-Tsung Lin. Cspot: Portable, multi-scale functions-as-a-service for iot. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 236–249, 2019.

[183] Rich Wolski, Chandra Krintz, Fatih Bakir, Gareth George, and Wei-Tsung Lin. Cspot: Portable, multi-scale functions-as-a-service for iot. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, SEC '19, page 236–249, New York, NY, USA, 2019. Association for Computing Machinery.

[184] Rich Wolski, Chandra Krintz, Fatih Bakir, Gareth George, and Wei-Tsung Lin. Cspot: portable, multi-scale functions-as-a-service for iot. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, page 236–249. ACM, Nov 2019.

[185] Explore the digi xbee ecosystem, 2021. `https://www.digi.com/xbee`.

[186] Ronghua Xu, Yu Chen, Erik Blasch, and Genshe Chen. Blendcac: A blockchain-enabled decentralized capability-based access control for iots. In *2018 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)*, pages 1027–1034, 2018.

[187] Ronghua Xu, Yu Chen, Erik Blasch, and Genshe Chen. A federated capability-based access control mechanism for internet of things (iots). In *Sensors and Systems for Space Applications XI*, volume 10641, 2018.

[188] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3), September 2002.