CSPOT: Portable, Multi-scale Functions-as-a-Service for IoT

Rich Wolski, Chandra Krintz, Fatih Bakir, Gareth George, and Wei-Tsung Lin
{rich,ckrintz,bakir,gareth,weitsung}@cs.ucsb.edu
Computer Science Dept.
Univ. of California, Santa Barbara

ABSTRACT

In this paper, we present *CSPOT*, a distributed runtime system implementing a functions-as-service (FaaS) programming model for the "Internet of Things" (IoT). With FaaS, developers express arbitrary computations as simple functions that are automatically invoked and managed by a cloud platform in response to events. We extend this FaaS model so that it is suitable for use in all tiers of scale for IoT – sensors, edge devices, and cloud – to facilitate robust, portable, and low-latency IoT application development and deployment.

To enable this, we combine the use of Linux containers and namespaces for isolation and portability, an append-only object store for robust persistence, and a causal event log for triggering functions and tracking event dependencies. We present the design and implementation of *CSPOT*, detail its abstractions and APIs, and overview examples of its use. We empirically evaluate the performance of *CSPOT* using different devices and applications and find that it implements function invocation with significantly lower latency than other FaaS offerings, while providing portability across tiers and similar data durability characteristics.

CCS CONCEPTS

• **Applied computing** → **Event-driven architectures**; • **Computing methodologies** → *Distributed computing methodologies*; • **Computer systems organization** → *Distributed architectures*;

KEYWORDS

IoT, serverless, cloud functions, append-only, portability, distributed systems

ACM Reference Format:
Rich Wolski, Chandra Krintz, Fatih Bakir, Gareth George, and Wei-Tsung Lin. 2019. *CSPOT*: Portable, Multi-scale Functions-as-a-Service for IoT. In *Proceedings of ACM Symposium on Edge Computing (SEC 2019).* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3318216.3363314

1 INTRODUCTION

The Internet of Things (IoT) is a rapidly emerging set of technologies that is fueling remarkable innovation in which ordinary physical objects in our environment are increasingly equipped with Internet connectivity, sensing, control, and computing capabilities. IoT application designers commonly combine these devices with the scale, services, and cost-effectiveness of cloud computing to implement IoT applications. However, at present, the heterogeneous (in terms of hardware, software, and APIs), asynchronous, highly scalable, dynamically changing, and geographically distributed nature of IoT-cloud applications, makes their infrastructure complex and difficult to provision, program, and optimize for high performance, energy efficiency, and scale.

One approach to taming this complexity goes by the moniker of "Serverless computing" (also known as Functions-as-a-Service (FaaS)) [6, 7, 17, 41] [1]. Amazon Web Services (AWS) released the first commercially available FaaS in 2014 called AWS Lambda [7]. The platform was originally developed to enable inexpensive and significantly simpler development and deployment of scalable, request-triggered web services and microservices [8, 9]. Given its success to date, other public cloud providers and open source communities have released FaaS platforms with similar functionality [13, 39, 45, 47].

FaaS developers structure their applications as simple, transient, stateless functions that access cloud services for their functionality. They upload them to a FaaS platform, which invokes the functions in response to specified system-wide events (e.g. storage updates, API requests, notifications, messages received, custom events, etc.). To do so, the platform automatically configures and provisions isolated execution environments (e.g. Linux containers [83]) on-demand. Users pay only for the resources and services their functions use during execution (and not for multiple virtualized servers). As a result, a FaaS web service is often easier to develop and operate. Because Linux containers typically provision more rapidly than virtual machines, FaaS applications also typically leverage autoscaling more efficiently than their virtualized server counterparts.

The event-driven programming style, fine-grained costing, easy integration of cloud services, and support for multiple high-level languages, makes FaaS attractive to IoT developers who plan to use a cloud to host their application backends. IoT applications are often event-driven (devices trigger computation, communication, and storage events), have unpredictable execution profiles (requiring efficient autoscaling), and perform a variety of data processing and analytics (which clouds increasingly export as managed services). To capitalize on this interest and potential fit of FaaS for IoT, and to reduce the response latency associated with using the public cloud, public cloud providers have begun to offer restricted versions of their FaaS platforms for "edge" servers and devices, i.e. remote compute and storage resources located near and directly connected

SEC 2019, November 7–9, 2019, Arlington, VA, USA

ACM ISBN 978-1-4503-6733-2/19/11…$15.00
https://doi.org/10.1145/3318216.3363314

[1] FaaS is synonymous with the term "serverless" because the FaaS platform automatically configures and manages the execution environment of functions and links them to cloud services, precluding the need for FaaS developers to provision servers explicitly to do so.

to IoT devices and sensors [24, 32, 35, 81, 87]. Example public cloud offerings include AWS Greengrass [2, 4] and Azure Iot Edge [65, 66]

To use these edge systems however, IoT applications must overcome multiple technology impediments not common to cloud-hosted web services. Specifically, IoT applications must be able to operate across a wide spectrum of computing devices, capabilities, and scales, from simple microcontrollers with kilobytes of memory and no virtualization, to privately-managed data centers, to public clouds supporting hundreds of different APIs and services. In addition, existing FaaS-based edge platforms (including AWS Greengrass and Azure IoT Edge), limit what operations are performed at the edge, fail when the edge is unable to reach the public cloud (a common scenario in IoT), and require that applications integrate a number of disparate programming models (e.g. publish-subscribe, embedded OSs, cloud SDKs, and FaaS) [10, 20]. Finally, as Hellerstein et al point out, FaaS itself is not a distributed computing technology, lacking a messaging capability and any way to exploit data locality [41] which IoT applications increasingly require. Thus, while FaaS is attractive from a cost and efficiency perspective, it currently does little to ameliorate other complexities facing reliable and pervasive IoT application deployment.

At the same time, low-power microntroller-based devices are becoming multi-functional. As a result, devices can now host services directly (rather than relying on cloud-hosted or edge-hosted proxies) leading to a new "Devices-as-services" architecture for IoT [14]. Under this approach (unlike current IoT-cloud approaches) it should be possible to write a single service once and then host it in the cloud, on the edge, or on a device, without modification. That is, resource capacity (and not programming heterogeneity and complexity) dictate hosting decisions exclusively in a Devices-as-services architecture. However, at present, no single runtime system or service implementation is capable of hosting services at device scale, edge scale, and cloud scale.

In this paper, we investigate an alternative approach for bringing FaaS to IoT that attempts to address these challenges. In particular, we design and develop a new IoT programming model (based on FaaS) and a distributed runtime system that enables code portability across all scales of IoT (sensors, edge, and cloud). The system, called *CSPOT*, also integrates a number of features that enable efficient, low-latency execution across IoT tiers, application robustness, AWS Lambda compatibility, capability-based security, and record and replay for application debugging and analysis [2].

The research contributions that we make to enable this approach are summarized as follows.

- We explore the feasibility of a single FaaS for IoT programming that makes multi-language program components portable across *all* platform scales – from microcontroller devices through the edge to the public clouds – in an IoT setting. Thus, in a *CSPOT* deployment, it is possible to move the computation to the data, or the data to the computation (which ever is most efficient) without recoding the application or its constituent software components.
- We extend FaaS with support for geo-distributed execution and dependency tracking. *CSPOT* storage is append-only (i.e. versioned) [52] for data durability and robustness in a distributed execution setting. No *CSPOT* function can be triggered without the generation of some concomitant datum in persistent storage. In addition, *CSPOT* runtime system is log-based [79], and all log records carry an identifier specifying the event that triggered the function. These features together make it possible to determine causal ordering of events efficiently and in a way that does not rely on statistical sampling. For scalable FaaS programs, particularly in an IoT setting where there may be many hundreds or thousands of sensors and actuators operating asynchronously and, thereby triggering a wide variety of analytical and control computations, determining the root cause of an error or unexpected program state is critical. The *CSPOT* abstractions are designed with this capability in mind.
- We define new FaaS abstractions for messaging and locality. Automatically garbage-collected, append-only storage objects (called WooFs) are addressed by Universal Resource Identifiers (URIs) and any access to a WooF across locality regions (called namespaces) is via a message. The combination of distributed, append-only storage and global causal event tracking makes it possible to implement useful debugging and data repair capabilities via targeted execution replay.
- We leverage emerging cloud and operating system technologies to enable isolated portability and low latency execution (key for near real-time, data-driven actuation, control, and automation at the edge). *CSPOT* couples Linux containers [83] for isolation (where available) with memory-mapped storage to support application data structures. The result is that *CSPOT* can dispatch isolated FaaS functions with latencies that are two orders of magnitude lower than current commercial cloud offerings.

We empirically evaluate *CSPOT* using a wide range of devices and performance metrics. In addition, we evaluate (i) persistent data repair and function replay, (ii) AWS Lambda and S3 Python compatibility, and (iii) capability-based security. Moreover, we compare *CSPOT* performance to that of existing, production-quality, alternatives: AWS Lambda and Greengrass and Microsoft Azure IoT Edge. Our results show that, relative to extant FaaS systems, *CSPOT* improves the response time and facilitates low-overhead end-to-end performance for IoT benchmarks and applications across IoT tiers/scales. To our knowledge, no other technology has taken the *CSPOT* approach of bringing the cloud to IoT rather than IoT to the cloud. As a result, our work addresses many of the complexity, performance (in terms of latency), and forensic (in terms of causal dependency tracking) challenges not addressed by alternative approaches.

2 *CSPOT* ABSTRACTIONS

Our *CSPOT* design is motivated by several observations that we have made while building and deploying IoT applications "in the wild". These applications are long-lived, operate in remote, unattended locations, and perform sensing, data production, processing, and analysis, and data-driven operations for surrounding systems. Many of the underlying devices rely on battery power or alternative/intermittent energy sources, have restricted or intermittent access to computational and/or network infrastructure, and experience failures (hardware and software) that result in data loss and corruption. To extend their capability, longevity, and robustness, the applications increasingly rely on more capable, co-located edge

[2] *CSPOT* is available as open source from https://github.com/MAYHEM-Lab/cspot.git.

systems for computational, communications, storage, and analytics offloading as well as for other proxy services. Low-latency access to these services is critical to enable the near real-time, data-driven alerting, actuation, control, and automation of physical, mechanical, and digital systems that these applications perform.

We find that most of these applications are modular and event-driven (i.e. operations execute in response to state changes and data arrival), making them suitable to the FaaS programming and execution model. However, FaaS systems today are not distributed, restrict both the type and scale of applications, and provide no automated, end-to-end error tracking or failure recovery. Further, they provide limited security mechanisms (if any) and incorporate heterogeneous devices (edge, sensors, microcontrollers) using disparate technologies (different operating systems, protocols, software development kits (SDKs), and FaaS services) – making them challenging to program, deploy, debug, and maintain. Extant FaaS systems for IoT also require that functions communicate global state through remote, location-concealed data services. This limitation prevents intelligent code placement and the exploitation of data locality (i.e. moving the code to the data and/or the data to the code) and becomes a bottleneck that limits performance and scale [16, 40, 41, 49].

CSPOT addresses these limitations via a portable, high-performance, robust runtime system capable of running on all IoT tiers – from sensors to public clouds – so that code and data can move between these infrastructures without the need for transformation or recoding. We also design *CSPOT* to be extensible to efficiently support key IoT application services, including causal event tracking, data repair, function replay, device/service access control, and API-compatibility with popular cloud offerings. To enable this, *CSPOT* defines, combines, and exports three abstractions:

- **Wide Area Objects of Functions** (WooFs), which are append-only memory objects with which developers persist state,
- **namespaces**, which root separate, hierarchical, declarative regions that provide a scope for WooFs, and
- **event handlers**, which developers use to define functions that are triggered when a data item is appended to a WooF.

The *CSPOT* programming model is event-driven; events are triggered by state updates to WooFs. The only way to persist data beyond handler execution is via WooF updates. Thus a *CSPOT* application consists of event-triggered computations that may generate volatile local state but that result in updates to application "variables", which are global to the application, have append-only semantics (i.e. multiple versions), and are persistent.

This "insistence on persistence" with versioning is motivated by the observation that at the device tier (and to a lesser extent the edge tier), computation, communication, and storage capabilities are far less reliable than in less hostile environments. Thus, *CSPOT* abstractions mandate that data persist after *all* meaningful computations so that it can be processed when temporary outages have been resolved. Our work focuses on making (as discussed in Section 3 more completely) FaaS persistence and communication "fast" and to ensure low latency response,application scalability, and effective management of scarce resources. *CSPOT* uses append-only semantics for its persistent data structures and distributed log to handle eventual consistency robustly and efficiently as is done in other distributed systems (e.g. HDFS, [34], SafeStore [52], Chariots [73], Corfu [15], etc.), by prioritizing high availability and partition tolerance over consistency [25].

Also unique to *CSPOT*, each namespace has a location (e.g. a host machine), which gives application developers control over code and data locality. Developers currently define namespaces via Universal Resource Identifiers (URIs) which map to IP addresses (we currently do not use DNS (but can) given its infrequent use in the remote IoT deployments that we consider). URIs (and WooF names) are posted to a web site for lookup and use by clients (we are investigating integration of a discovery service in *CSPOT* as part of future work).

Each WooF and handler identifies uniquely with a *CSPOT* namespace and namespaces cannot overlap. Each handler operates directly on WooFs within its namespace only. Handlers are triggered when a WooF is appended to by a handler. That is, *CSPOT* couples FaaS function invocation with persistent storage in a 1-to-1 correspondence to aid debugging, profiling, and management of highly concurrent IoT applications.

Communication between namespaces is performed via point-to-point messages (direct socket connections) sent by a handler in one namespace performing an update on a WooF encapsulated in another namespace. This mechanism also makes it possible for WooFs to be accessed (and thus to trigger computations) from external processes, including non-*CSPOT* programs and services.

Within each namespace, *CSPOT* maintains an append-only, event log of WooF updates. *CSPOT* uses this event log to trigger handlers and to track causal dependencies within a *CSPOT* application (across handlers). Causal dependencies can be used to facilitate data replication, synchronization, root cause analysis, and replay by *CSPOT*. The size of the event log is a tunable parameter.

Each WooF is logically also a log of append operations where each element appended is an untyped memory region of fixed size. The element size is set when a WooF is created. Each WooF append returns a unique sequence number associated with the appended element. All elements between the element most recently appended and the "earliest" element in the history can be accessed (i.e. there are no missing elements between elements that are present in the WooF history). Thus *CSPOT* maintains a version history of fixed-size for each persistent data structure, and each version is identified via a unique ID. *CSPOT*'s append-only semantics make data structures logically immutable [28, 52, 62, 79, 80] facilitating, among other benefits, data durability, access/update efficiency, version control, debugging and repair, and lineage/provenance tracking. *CSPOT* garbage collects WooFs and logs by overwriting the oldest elements (i.e. using a circular buffer).

The *CSPOT* API

The *CSPOT* API consists of a `create` operation that defines the name, element size, and history length for each WooF, a `put` operation that appends an element to a WooF (returning its sequence number), and a `get` operation that returns the element corresponding to a sequence number. The API also includes operations that allow a programmer to get the attributes associated with a WooF, to delete a WooF, and to create and destroy namespaces and handler resources. We omit the details of these latter operations for brevity but document them in the open source repository.

WooFCreate() creates a WooF in the local namespace.

WooFPut() causes the untyped memory pointed to by one of its parameters to be appended to the specified WooF (i.e. it is a "blob" store [54, 82] in which each blob has a fixed size). When **WooFPut()** succeeds, the unique sequence number assigned to the appended element is returned to the caller.

WooFPut() also takes the name of a handler as an optional parameter that is the name of a handler binary located in the same namespace as the WooF that *CSPOT* will invoke once the append has been successfully completed. Thus data is always appended to a WooF, but the programmer can determine when a handler should be triggered as a result of the append. However there is no facility for invoking a handler without appending to a WooF. In this way *CSPOT* maintains a 1-to-1 mapping of data persistence to computation, i.e. every function invocation is unambiguously caused by a particular WooF update (identified by sequence number).

A call to **WooFGet()** returns the element corresponding to the sequence number that is optionally passed in as an argument (the default is the latest version, i.e. the WooF tail). Note, again, that the size of the memory region that **WooFGet()** writes is determined by the element size specified when the WooF is created.

Handlers and Clients

The *CSPOT* API can be invoked either within handlers or from "client" programs that are interacting with one or more *CSPOT* applications. Each handler must have the following C-language prototype. However, as we show in Section 6, we support handlers written in other languages (e.g. Python) via C bindings. Other high-level programming languages (e.g. those with managed runtimes) can be supported in a similar way and we are considering such extensions as part of future work.

*int handler_function_name(WOOF *wf, ulong seq_no, void *ptr);*

`handler_function_name` is a legal C-language function name (i.e. the handler that will be compiled as a C-language function having these three arguments). The first parameter is a C-language pointer to a structure defined by *CSPOT* for manipulating WooFs. The second parameter is the sequence number that *CSPOT* assigned to the WooF append. Note that the append occurs prior to handler invocation. The third parameter is an "in" pointer that points to a copy of untyped memory that has been appended. This memory is logically immutable so a change by the handler to the memory pointed to by the *ptr* function does not persist beyond the handler's execution lifetime. By convention, handlers return zero on success and a negative value on failure.

In the current version of *CSPOT*, handlers should only persist state via a call to **WooFPut()**. However, this restriction is not currently enforced so it is possible for them to make network connections to access outside services, some of which could also persist application state.

Clients are programs written in any programming language that use the *CSPOT* API. Logically, a call to a *CSPOT* API function from a program that is not a handler results in a request to the WooF's namespace to perform the operation on behalf of the caller and to return the results.

3 *CSPOT* IMPLEMENTATION

We implement *CSPOT* v1.0 using Linux memory-mapped files [84] as the operating system storage abstraction for WooFs on all devices that support Linux. We isolate function handlers as Linux processes executing within a Docker v18.09 [27] container associated with each namespace (each serviced by 1+ containers). Handler execution constitute "events" within the system. *CSPOT* triggers handler execution via its event log. Autoscaling is controlled by throttling invocations using a maximum concurrency level per namespace (without prioritization). Developers can query the log to extract the causal order of all events within a namespace for use as a debugging aid. For cross namespace invocation, we use ZeroMQ [89] as the messaging substrate and threads within the container to proxy namespace-external operations.

WooFs

Each WooF is implemented as a separate memory-mapped Linux file containing a typed header structure and space to contain some number of fixed-sized elements. The header includes the local file name of the WooF, element size, the number of elements that are retained in the append history, and the current sequence number.

Each WooF keeps a circular buffer of appended elements. The head and tail indices are stored in the header. The space for the buffer is located immediately after the header in the memory-mapped file.

Thus WooFs are self-describing in that all of the information necessary to manipulate a WooF are contained in the WooF itself. When a WooF is "opened", its contents are mapped into the memory space of the process opening the WooF as shared memory. Thus multiple threads and, indeed, multiple processes can access a WooF concurrently using the information contained in the WooF header.

To implement synchronization for internal operations, the WooF header includes two Linux semaphores [59]. The first implements mutual exclusion for operations like buffer head and tail index update, sequence number assignments, etc. The second allows threads to synchronize on the "tail" of the WooF so that when a new append occurs, they can be activated. These semaphores are not exposed through the *CSPOT* API, however, and are used strictly by the runtime.

Handlers, Containers, and the Event Log

When *CSPOT* starts a namespace, it launches a Docker container for the namespace, which shares the namespace directory in which all WooFs and handlers for the namespace are located. Docker includes an option to specify where, in the container's directory structure, a directory shared with the host must be located. By using the same location within the container (e.g. "/*CSPOT* ") the API can locate the WooF and handlers within the container.

Each handler is compiled as a separate Linux executable program. When an invocation of **WooFPut()** includes a handler name, the API code appends the element specified in the call to **WooFPut()** to the WooF and then appends an event record specifying the WooF, the sequence number of the element that has been appended, and the handler name to the *CSPOT* event log for the namespace.

The main process within the container spawns several threads that synchronize on the tail of the event log in the namespace using a semaphore in the event log header. These threads "claim" events from the log by atomically appending a claim record for an unclaimed event. They then call Linux `fork()` and `exec()` on the handler binary. When **WooFPut()** is called from within a handler,

the sequence number of the caller is included in the event record indicating that it is the "cause" of the handler firing. Thus the event log that serves as the dispatch mechanism for handlers, also includes the dependency information necessary to determine causal order.

To determine a global causal ordering, *CSPOT* includes a log-merge tool chain that combines event logs from multiple namespaces. It creates a single total order of events in which the causal dependencies, both within and across namespaces, are correctly represented. Multiple log merges produce the same causal order, but may produce different total orders depending on namespace merge order (unlike systems designed to produce a single total order across logs [15, 73, 88] which is more costly).

CSPOT for Microcontrollers

Because microcontrollers typically do not support virtual memory, the implementation of *CSPOT* for these systems does not rely on memory mapped storage. Instead, we have developed a multitasking system in which memory is addressed directly, but computations can still be run in threads. The system uses `setjmp` and `longjmp` primitives for stack manipulation. The *CSPOT* microcontroller system is currently portable across the ARM, AVR, and Espressif microcontroller processors.

The *CSPOT* namespace server for microcontrollers implements the same API semantics of the Linux implementation. However, all handlers executing within a microcontroller are part of the same trust domain due to a lack of memory isolation. We execute handlers on different runtime stacks for resource isolation purposes.

A portion of the flash memory is used as the backing storage for WooFs and namespace logs. To preserve the lifetime of the backing flash memory, contents of WooFs are written back to non-volatile storage at most every 2 seconds. As the controller also lacks a DMA controller and the flash can only update whole sectors, the delayed write back also benefits performance.

Finally, to save both memory space and porting effort, we implement cross-namespace messaging using the msgpack protocol [68]. Msgpack is similar in its data framing to ZeroMQ, and we implement the minimal functionality required for consistent messaging.

4 *CSPOT* SECURITY

For device authentication, we have developed capability-based security for *CSPOT* . Our approach is inspired by Macaroons [23] but specifically designed for low-power device and edge implementations. Because *CSPOT* must run at all scales, and the security protocol is intrinsic to its function, we have developed this mechanism to be computationally efficient, space efficient, and to allow secure delegation of access policies in a way that does not require coordination messages between the principal and the delegate. As a result, the *CSPOT* capability mechanism is suitable for implementation on embedded systems and microcontrollers, where computational load, radio activation, and storage usage must often be minimized to save battery power.

Because TLS [78] and other public-key encryption systems are computationally and space expensive, our system uses a capability derivation [70] mechanism based on HMAC [53] cryptographic hashing. When a device is commissioned (added to a deployment through a manual registration process), we furnish it with a secret "nonce" to use in generating digital signatures using HMAC.

Each time a resource is created (WooF, namespace, or handler) the creating device creates and signs a capability by hashing the capability and the nonce together. The capability describes the access rights the holder of the capability (user) is entitled to exercise. Each *CSPOT* message that operates initiated by the user on a resource must include a capability that indicates that the sender of the message is entitled to perform the operation. The message receiver checks the validity of the capability by appending the nonce to the capability, hashing the pair, and comparing it to the signature carried in the capability. If the nonce remains secret, it is cryptographically unlikely that the capability has been forged.

Thus the capability issuer (called the "server" in *CSPOT* parlance), which can be any *CSPOT* device including a microcontroller, can verify any capability without storing a copy of it. Further, *CSPOT* supports complex access policies using a "chain" of capability derivations constructed via capability attenuations.

Specifically, *CSPOT* crafts a capability as part of the create API for each resource type (namespace, WooF, and handler) granting all possible access rights for the resource instance. *CSPOT* returns the capability upon successful completion of the create operation to the entity requesting creation of the resource (e.g. a *CSPOT* handler or client). The entity then includes the capability in all subsequent API calls (resource access requests, e.g. WooFPuts and WooFGets).

Any entity holding the capability can attenuate (i.e. "lower" the permissions) by adding a new less powerful capability to the end of a chain that has the root capability as its first element. It adds an attenuation and cryptographically hashes the chain along with the signature associated with the chain that does not include the attenuation being added. It then replaces the signature for the chain (including the new attenuation) with this hash. That is, the chain only carries the last signature which comprises all previous signatures but obscures them cryptographically. As a result, only the holder of the nonce can reproduce the correct sequence of signatures starting at the head of the chain.

When *CSPOT* is presented with a chain of capability attenuations, it rehashes the root with the nonce to generate the signature that was used from the root in conjunction with the first attenuation to generate the signature carried with the first attenuation. It then repeats the process (using the "current" signature and the next attenuation to generate the next signature) until it reaches the end of the chain. If at the end, the signatures match, the holder of the nonce can assert that each successive attenuation (starting from the root) is valid.

Thus it is possible for a *CSPOT* device to verify a chain of unforgeable attenuations that has been correctly aggregated by *any* set of participating devices. In particular, if the device and a delegated policy engine exchange secrets during commissioning so that the device and the engine can mutually authenticate, then the policy engine can issue and manage attenuations on behalf of the device that the device can always perform verification independently.

We perform two optimizations in this scheme. First, to prevent repeated hashing operations associated with long chains, the device can "squash" a chain by simply issuing a separate capability carrying the the permissions in the last attenuation. Secondly, it is possible to merge separate HMAC hashes in a way that permits fast verification. Thus it is possible to create attenuations for sets of capabilities that were generated separately, thereby permitting

arbitrary policy implementation. We plan to explore enriching this security protocol and its optimization as part of future work.

Note that the current *CSPOT* security mechanisms do not address privacy intrinsically. In particular, *CSPOT* capabilities cannot be forged but they can be stolen and the current *CSPOT* system does not implement a notion of identity. We are also investigating these important issues also as part of future work.

5 AN EXAMPLE *CSPOT* APPLICATION

We use *CSPOT* as part of an IoT system for agriculture, which aids growers and farm managers in frost protection for crops. One method of frost prevention uses large wind-generating fans to mix warm air aloft with cold air near the ground. The fans are typically propane or diesel powered, causing considerable expense (in terms of fuel cost) and carbon emissions when they are in use. Thus farmers would like to know, with a considerable degree of accuracy, the temperature differential between the air at approximately 10 meters altitude and at 1 meter altitude and for many locations (microclimates) in their growing blocks, so that they can more precisely control fan use.

Current solutions to this problem rely on manual labor to drive though the property reading fixed thermometers. This solution is error prone and expensive since the labor force must work through the night when frost is likely (at least in the locations we study).

Our IoT system consists of inexpensive, low power temperature sensors deployed widely and at multiple altitudes across the farm. The devices measure temperature information in real time, analyze it for the temperature gradients that indicate fan activation is necessary, and monitor the temperature change caused by fan activation to ensure efficacy.

Currently, at each measurement location, we deploy a Raspberry Pi Zero with an attached temperature and humidity sensor. This installation uses a battery and a small set of solar panels that charge the battery during the day so that the device can run at night. During the course of this project, we observed that the internal CPU temperature, as reported by the on-board health-and-status interfaces implemented by Raspbian [77] (a Linux variant for the Raspberry Pi platform), is highly correlated with outdoor temperature.

Figure 1 shows time series traces for the outdoor temperature (as measured by a commercial-grade meteorological station) and the internal CPU temperature for a Raspberry Pi "Zero" [76] located at one of our farm locations. The meteorological station measures outdoor temperature at 10 meters and the Raspberry Pi (located in a weatherproof container) is at a 1 meter altitude.

From the Figure (and a number of other experiments including those that use commercially available meteorological data), it is clear that outdoor temperature can be predicted from CPU temperature. However, note that there are some discrepancies in shape between the two curves. To generate an accurate prediction of outdoor temperature, our IoT application smooths the CPU series using Singular Spectrum Analysis (SSA [38]) and computes a linear regression between the smoothed CPU series and the observed outdoor temperature. SSA requires a number of lags of autocorrelation to use and a finite history. In Figure 1 the system chooses up to 12 lags (30 minutes) over a history of 24 measurements (2 hours). It recomputes both the smoothed series and the regression coefficients

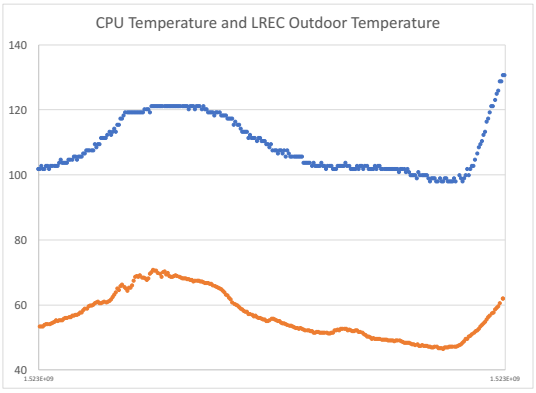

Figure 1: Time series trace of outdoor temperature and CPU temperature taken from an IoT deployment at our experimental farm. The units of the y-axis are degrees Fahrenheit. The sensor generates a measurement every 5 minutes.

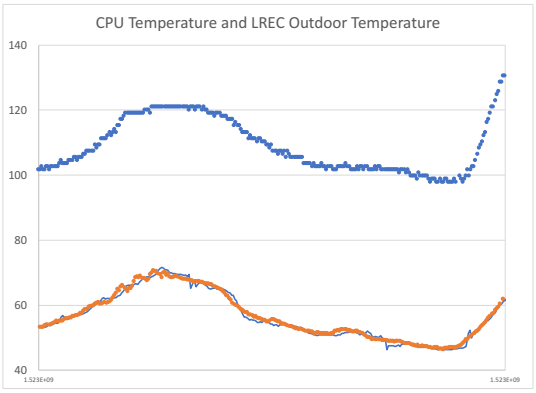

Figure 2: Time series trace of outdoor temperature, CPU temperature, and predicted outdoor temperature taken from an IoT deployment at our experimental farm. The units of the y-axis are degrees Fahrenheit. The sensor generates a measurement every 5 minutes.

using *CSPOT* handlers every time a new outdoor measurement is posted to a WooF. Similarly, every time a new CPU temperature measurement is posted, it uses the most recently computed regression coefficients to predict the outdoor temperature.

Figure 2 shows all three series: the CPU temperature series, the outdoor measurement series, and the predicted outdoor temperature series. In this figure, the measurements are shown as individual markers and the solid line shows the predictions. Despite some obvious deviation, over this time period (24 hours), the mean absolute error between the measured outdoor temperature and the predicted outdoor temperature is 0.73 degrees Fahrenheit with a standard deviation of 0.61. We have shown in other work that this application achieves average absolute errors of less than 1.5 degrees Fahrenheit for a wide ranges of settings and devices, even when under load (compute and transfer) [37]. By obviating the need to fit the Pi with an external temperature sensor, we reduce power consumption and

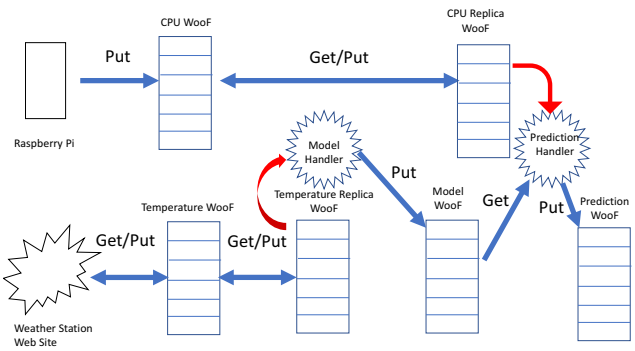

Figure 3: *CSPOT* Frost Prevention Application Structure

free up ports on the device for use by other sensors, e.g. to measure other atmospheric phenomena, soil temperature and moisture, etc.

Figure 3 shows the structure of this *CSPOT* application. The Raspberry Pi calls **WooFPut()** every time a CPU temperature measurement is to be taken (every 5 minutes for the data in Figure 2). Similarly, an agent that is polling a web site where the weather station posts its data, executes a **WooFPut()** to store a raw outdoor temperature measurement (every 5 minutes).

For data durability and overall robustness of the application, we relay this initial data to two intermediary client applications running on a wall-powered, edge device on-farm, which fetch the data using **WooFGet()** and then forward it via **WooFPut()**. Note that these intermediaries can query both source and target WooFs to ensure that all sequence numbers are effectively forwarded.

The puts to the secondary data replica WooFs on the edge device trigger a separate handler function. When data is put to the CPU temperature replica WooF, the handler fetches the latest model that has been fit using the most recent outdoor temperature data and makes a prediction of outdoor temperature that it puts to the prediction WooF. When new outdoor temperature data is put to the temperature replication WooF, a triggered handler computes a new prediction model using the latest outdoor temperature and latest CPU temperature data. The model parameters are put to a model WooF where they can be fetched by the CPU measurement handler that is triggered for each new value appended to the CPU replica WooF.

This application architecture is one of several different possible architectures that would produce equivalent results. In particular, it is possible to have puts to the initial data WooFs trigger model generation and temperature prediction directly (i.e. without the use intermediary clients and replica WooFs). *CSPOT* enables developers to easily explore these alternatives in their applications.

However, from a deployment perspective, this architecture offers several advantages. First, the sensors are located in a remote location where network connectivity is both low quality and power intensive over long distances. Thus, the first level WooFs are hosted in an out-building near the sensors on an edge cloud that communicates with the sensors via a local, isolated network.

The replica WooFs are hosted in a data center cloud that also runs the intermediary client application components. The advantage of this approach is that the edge cloud need not take responsibility for delivering data to the data center cloud, thereby freeing cloud resources to maximize the chance of correct data acquisition in the face of a lossy network. Also, by replicating the data in the data center cloud, it is possible for the edge cloud and the data center cloud to operate independently, the latter using data that is eventually-consistent. In this way, loss of network connectivity to the edge cloud allows an outdoor temperature prediction using out-of-date temperature information. Because outdoor temperature does not fluctuate significantly on a 5 minute time scale, network outages of a relatively short duration do not cause the application to have to "fail stop."

Note that this deployment architecture is not built into the application itself. That is, the entire application can be hosted on the edge cloud or (if the sensors can communicate with the wide-area Internet) in a data center cloud. Note also that the intermediary clients can be removed transparently. That is, if the WooFs at the edge do puts in the handlers that are triggered, the "back end" of the application does not need to change. Thus application structure can be altered to fit different deployment, reliability, and power-usage requirements while the application, itself, remains unchanged.

6 EVALUATION

We next empirically evaluate the performance profile of *CSPOT* across a spectrum of scales using benchmarks and end-to-end IoT deployments. We compare to *CSPOT* to FaaS-based edge systems AWS Lambda (using *CSPOT* functions written in C and Python) and Azure IoT Edge. We also investigate *CSPOT* extensibility via novel IoT services for function replay and data repair. We then evaluate the overhead of making *CSPOT* API-compatible with AWS Lambda and AWS S3 (the Simple Storage Service [11]) so that any AWS Lambda function written in Python (using S3 as its event source) can execute over *CSPOT* without modification. Finally, we assess and report on the performance impact of adding capability-based security to *CSPOT*, combining virtualization mechanisms, and *CSPOT* reliability.

To evaluate *CSPOT* performance, we benchmark the runtime system across the hosts shown in Table 1. We synchronize each host clock using `ntp` and the same time server for all experiments. The deployment includes devices (sensors and single board computers), an edge cloud, a private cloud, and a public cloud. The devices are a microcontroller (esp8266) [30] and a Raspberry Pi Zero W (RPi0) [76]; the edge cloud is an Intel NUC (NUC) [46]. All are connected via WiFi. The edge cloud (the Intel NUC) and private cloud runs Eucalyptus version 4.3 [31, 74]. We use an *m5.xlarge* instance in AWS EC2 [72] in the *us-east-1* region for the public cloud component.

The private cloud is at UCSB in a secure and maintained data center, is connected to the Internet via a layer-3 IP network, and is used in production by approximately 1000 users per academic quarter. We refer to it as the *Campus private cloud* in the tables. We did not quiescent or otherwise isolate this private cloud infrastructure.

We deploy *CSPOT* and the benchmarks described herein on each of these systems. Note that we use *the same source code* to compile the benchmark binaries for each host. Thus the benchmarks detail portable performance across all scales in a tiered IoT-cloud setting.

Host	Make and Model	CPU	Memory	OS
device	Espressif esp8266	80 MHz L106 RISC	112 KB	*CSPOT* native
RPi0	Raspberry Pi Zero W	1 GHz single Core BCM2835	512 MB	2018-06-27 Raspbian Stretch Lite
NUC	Intel NUC 6i7KYK	2.6 GHz Intel Core i7-6770HQ	32 GB	CentOS 7.2
Campus private cloud	cg1.4xlarge	2.1 GHz Xeon, 4 vCPU	8 GB	CentOS 7.2
AWS EC2	m5.xlarge	2.5 GHz Xeon, 4 vCPU	16 GB	CentOS 7.5

Table 1: Edge cloud testbed for *CSPOT* performance benchmarking

Host	Mean (ms)	std-dev (ms)	95% (ms)
esp8266	0.13, 38	0.015, 1.2	0.16, 40
RPi 0	37	6.8	48
NUC	4.0	0.63	4.9
NUC-VM	6.5	3.3	15
Campus private cloud	5.0	1.6	7.0
AWS EC2	5.0	0.96	6.6
AWS Lambda	253	90	584

Table 2: Comparison of *CSPOT* function dispatch times across different devices, edge, and cloud hosts. The statistics are each computed from 100 function invocations. The client and handler are written in C for all *CSPOT* results. The esp8266 results show statistics without and with flushing, respectively. For the AWS Lambda results, the handler is written in Python (C is not directly supported) and executed on AWS Lambda in response to a DynamoDB write. We evaluate a *CSPOT* Python handler later in this section.

We first benchmark *CSPOT* function invocation across tiers. Table 2 shows the mean, standard deviation, and 95^{th} percentile function invocation performance for 100 function invocations. In each data cell of the table, the units are milliseconds. The benchmark application runs a client on the same host where the WooF is located causing the *CSPOT* to bypass the ZeroMQ messaging layer. The client records the current time using the `gettimeofday()` system call and then executes **WooFPut()** to put this time stamp into a pre-installed WooF. The call to **WooFPut()** invokes a handler that reads the first time stamp, calls `gettimeofday()`, and writes both timestamps to a second WooF. After the experiment is complete, the benchmark reads the sequence of timestamp pairs from the second WooF. The difference between each pair of time stamps is the time for handler invocation by the *CSPOT* runtime. Both the client and handler are written in C.

Currently, *CSPOT* throttles handler invocation requests so that no more than 5 are active at the same time (which is a tunable parameter) per namespace container. In these experiments, we deploy a single container per namespace.

The table shows two values for the microcontroller (esp8266) results, separated by commas. Because the microcontroller does not implement virtual memory, the *CSPOT* implementation must copy the data from data memory to flash memory. To prevent undue wear on the flash, *CSPOT* performs this flush operation asynchronously, once every 2 seconds. Thus, the first number is the dispatch time for a put that does not synchronize with the flash memory and the second is the time necessary to synchronize the flash memory before the handler is fired. Note also that the microcontroller implementation does not support multiple namespaces per device. That is, all WooFs sited on a microcontroller must be part of the same namespace which is also sited on that microcontroller.

The table also includes two *CSPOT* edge deployments. NUC-VM is an edge cloud which uses KVM as a hypervisor to host the namespace and the client in a virtual machine instance running on an Intel NUC. NUC shows the performance *CSPOT* runtime natively on the NUC (i.e. not in a virtual machine).

For comparative purposes, we have also implemented the benchmark in Python (AWS Lambda currently does not support handlers written in C). The *AWS Lambda* results in the table use this benchmark and AWS Lambda for execution [7]. For this case, the handler is triggered by an update to DynamoDB [3] (i.e. DynamoDB is the event source). We evaluate benchmark performance for a handler written in Python (using *CSPOT*) later in this section.

The results in Table 2 indicate that *CSPOT* is able to achieve high rates of invocation performance regardless of the language used or the scale of the system on which it is deployed. Indeed, all x86 deployments (NUC, NUC-VM, Campus cloud, and AWS EC2) are *two orders of magnitude* faster than AWS Lambda. The microcontroller is three orders of magnitude faster when it does not flush WooF appends and one order or magnitude faster when it does. Finally, the Raspberry Pi (with its slower BCM processor, memory system, and storage) is one order of magnitude faster. In terms of latency, *CSPOT* represents a substantial performance improvement over AWS Lambda even when it is run in AWS EC2.

Also note that the NUC, campus private cloud, and EC2 invocation times are all similar. This is somewhat surprising given their very different internal engineering and scale. Curiously, the mean of the virtualized deployment on the NUC is higher than the others due to the presence of more values near the tail of the distribution. That is, in the virtualized case, the run time occasionally generates a much longer execution time (as evidenced by the larger 95^{th} percentile) than in the non-virtualized case. It is not possible for us to perform the virtualized versus non-virtualized experiment on the campus cloud or EC2. However, the 95^{th} percentile in each case is much closer to the mean than in the NUC-VM case, indicating that the effect is not as pronounced if present.

To make the comparison a fair one, for the Lambda benchmark, we use a "warm start". That is, we run the benchmark once to allow AWS to employ whatever state or container caching it might implement internally and then run the benchmark immediately thereafter. Note that *CSPOT* pre-allocates some number of containers for each namespace (each of which carries a throttle) upon deployment so there is no "cold start" penalty. Timing data is taken from the second run and for 100 invocations.

Host	Mean (ms)	std-dev (ms)	95% (ms)
NUC	16	0.57	17
NUC-VM	18	1.2	19
Campus private cloud	22	0.6	23
AWS EC2	18	3.1	23

Table 3: *CSPOT* function dispatch times for the edge and cloud hosts when the handler is written in Python. The statistics are each computed from 100 function invocations.

Note also that because *CSPOT* uses a memory-mapped, append-only log to trigger events, it captures captures timings and causal dependencies automatically. That is, the runtime event log is the "ground truth" for event invocations. In contrast, AWS Lambda (and FaaS equivalents from Microsoft and Google) rely on external sampling systems (e.g. AWS X-ray [12]) to capture event timing and dependencies. Further, *CSPOT* carries these dependencies across namespaces and, thus, across tiers and across clouds. Vendor FaaS systems are only able to track dependencies within the confines of each specific FaaS system – not across cloud services within a single cloud or across clouds.

One difference between the AWS Lambda results and the *CSPOT* results in Table 2 is that the Lambda benchmark is written in Python while the *CSPOT* implementation uses C. To measure the effect of this difference (C is widely believed to be significantly faster than Python) we have developed a simple Python binding for *CSPOT* handlers. Using this binding, we recoded the Lambda benchmark for *CSPOT* in Python (the *CSPOT* runtime is still written in C).

Table 3 shows the *CSPOT* timings for the benchmark when coded in Python using the x86 architectures. These results indicate that use of the Python programming language in *CSPOT*, introduces only 3–4x slowdown vs C for this benchmark. Thus, using Python, *CSPOT* is one order of magnitude faster than AWS Lambda. Generally, from the perspective of invocation latency, these results show that *CSPOT* is high performance and lightweight across a spectrum of platform scales.

6.1 Deployment Performance

While the data in the previous tables indicates that *CSPOT* is able to achieve high performance as a FaaS system, we also evaluate its performance within an IoT deployment. The application takes a local time stamp and calls **WooFPut()** on a remote WooF to store it. The handler for this WooF takes a local time stamp and calls **WooFPut()** on a remote handler, and so on, until the end of the "chain" of puts. In this way, the data is replicated on each WooF before it is forwarded to the next. Thus, this application implements "weak-chain" replication as described in [86] by van Renessee and Schneider but without the "master" process that is responsible for implementing chain reconfiguration.

Note that this communication pattern represents an alternative form of eventually consistent storage to that typically offered in most public clouds. Using the public clouds, data is transmitted from the device (possibly through an edge device) to the public cloud for storage, and that storage is replicated using eventual consistency. As a result, there is no possibility to exploit geographic locality. *CSPOT* allows the deployment designer to implement replication throughout the device-edge-cloud hierarchy in a way that enables both disconnected operation and locality-based optimizations.

Table 4 shows the end-to-end latencies in milliseconds for various *CSPOT* deployments. It also shows the same benchmark results when implemented using AWS Greengrass [2] and Microsoft Azure IoT Edge [65]. Greengrass is an AWS service designed to enable the AWS Lambda function execution in edge computing devices. Greengrass for IoT consists of two complementary technologies: The AWS IoT SDK [4] or FreeRTOS [19] for devices, and Greengrass "Core" which implements the AWS Lambda runtime (using Linux containers) for edge computing platforms. With Greengrass, a device (using the AWS IoT SDK or FreeRTOS) publishes its data via the MQTT [44] protocol.

In this experiment, we have implemented MQTT for the esp8266 according to the protocol specification in described in [63]. The Core (which is available for different edge computing platforms) subscribes to an MQTT event stream for the device and triggers a Lambda function when each event arrives. That function can then either invoke other Lambda functions locally within the core, or access other AWS services in the public AWS cloud. Note that AWS Lambda does not include a data persistence abstraction in the edge so data that persists must be stored using one of AWS's many storage services, all of which are available only as public cloud services. Thus, a Lambda function running in the Greengrass Core at the edge, can only persist data in the AWS public cloud and only *when* the edge and cloud are connected.

Azure IoT Edge also uses MQTT for data acquisition and can trigger a function when data is published. Like *CSPOT* (but unlike Greengrass), Azure IoT Edge includes a simple storage capability at the edge. However this storage capability (i.e. an SQL database) is not able to trigger arbitrary functions when updated. Further, at the time of the investigation we were unable to get the Python IoT Edge module to run our benchmark on the RPi0. Similarly, we were unable to integrate the client-side MQTT for the esp8266 with IoT Edge's server implementation with sufficient fidelity to run the benchmark. Thus, the IoT Edge timings are from the NUC-based edge cloud (*edge*) to Azure (the public cloud) where the data is persisted using CosmosDB [64].

The first four data rows of Table 4 show the latencies from different *CSPOT* deployments, each corresponding to a different strategy. In each case, the initial timestamp is generated on the microcontroller (*esp8266*). The *esp8266-WooF* strategy stores the timestamp on the microcontroller before forwarding the data to the next host in the "chain" (via a WooF handler).

For example, the row containing *esp8266-WooF->RPi0->edge->campus->AWS* stores 5 replicas of the data passed to the first call to **WooFPut()** in WooFs along the chain. The first is stored on the microcontroller, the second is on the RPi0, the third is on the edge cloud VM (NUC-VM in the previous tables), the fourth is on the campus private cloud, and the fifth is in AWS EC2.

Alternatively, rows prefixed with *esp8266* (where WooF is elided) show the benchmark configured with the *CSPOT* client executing a **WooFPut()** directly on a remotely hosted WooF (either *RPi0* or *edge VM* in the table). In each case, the second column shows the number of replicas that are generated.

The last two rows of the table show the results for a version of the benchmark executing in Greengrass and Azure IoT Edge,

Deployment	Replicas	Mean (ms)	std-dev (ms)	95% (ms)
esp8266-WooF->RPi0->edge->campus->AWS	5	535	61	650
esp8266->RPi0->edge->campus->AWS	4	513	48	607
esp8266-WooF->edge->AWS	3	298	14.6	326
esp8266->edge->campus->AWS	3	323	17	457
AWS Greengrass (esp8266->edge->AWS)	>= 3	4136	632	4288
Azure IoT Edge (edge->Azure)	>= 3	2621	1512	4386

Table 4: Comparison of *CSPOT*, Greengrass, and Azure IoT Edge deployments across the wide area. The statistics are computed from 100 consecutive runs. *esp8266* is the microcontroller, *edge* is NUC-VM, *campus* is the private cloud, and *AWS* is AWS EC2.

respectively. In these cases, because a cloud database is used for persistence, the public clouds will make at least 3 replicas of each datum. For reference, the current set of *CSPOT* applications that we have deployed (including on-farm) as part of IoT settings (approximately 100 of them), use the *esp8266->edge->campus->AWS* deployment configuration.

The performance of individual *CSPOT* abstractions is significantly faster (i.e. an order of magnitude) end-to-end than current IoT offerings from either AWS and Azure in these wide area deployments. Moreover, the distribution capabilities of *CSPOT* make it possible to develop replication strategies that are lower latency with similar data durability characteristics. Finally, *CSPOT* runs the same handler across all tiers precluding the need for recoding or for use and management of disparate programming models, libraries, and protocols that AWS and Azure require (e.g. FreeRTOS, SDKs, MQTT, FaaS, etc.).

6.2 Implementing Replay and Data Repair

We next evaluate the extensibility of *CSPOT* by using it to implement distributed function replay. Dependency tracking is useful in IoT settings to dynamically analyze, debug, simulate, and reason about distributed and highly concurrent (e.g. FaaS-like, event triggered) applications [1, 21, 22, 36, 48, 56, 60, 61]. Record and replay and data repair is useful in IoT settings to analyze and recover from data corruption *in situ* [55, 62]. Gathering data in one place repeatedly for analysis can consume significant power and resources, and is thus infeasible in some IoT settings. Repair and replay enables us to "move the analysis code to the data" instead of vice versa to save both time and resources for resource-constrained and disconnected IoT settings.

We refer to this *CSPOT* service as *SansSouci*. *SansSouci* scans WooF append histories and their causal relationships in the *CSPOT* log, and uses them to replay function invocations and repair data structure versions. Similar in spirit to Retroactive Lambda [62] but portable to all tiers, *SansSouci* is useful as a development and debugging tool for IoT applications and deployments. If a developer wishes to experiment with a new event-handler, she can install the handler, and replay the event stream (from the oldest values recorded in the application's WooFs) to observe the effect of the change. Restoring the old handler to reverse the experiment.

SansSouci uses specific causal dependencies (from the *CSPOT* runtime system logs) to execute only those computations that are needed to replace dependent data. For this reason, *SansSouci* can be significantly more efficient than back-up and recovery in the case of data corruption. For example, if a device fails and begins generating faulty telemetry, and the developer wishes to replace the corrupted values with some reasonable approximation (e.g. the average of some set of similar readings), *SansSouci* will only re-initiate the computations (events) downstream of the faulty device that are dependent on its telemetry for input.

	WooFPut	WooFGet
Cloud	143.42us (2.57us)	142.94us (3.51us)
Cloud w/ *SansSouci*	143.72us (3.55us)	142.76us (3.45us)
Cloud during repair	185.40us (2.51us)	157.89us (1.10us)
RPi3 (edge)	504.62us (3.92us)	521.20us (8.84us)
RPi3 (edge) w/ *SansSouci*	506.24us (10.44us)	523.14us (5.61us)
RPi3 (edge) during repair	681.91us (11.81us)	519.75us (4.61us)
esp8266	26.07us (0.17us)	23.45us (0.15us)
esp8266 w/ *SansSouci*	26.45us (0.17us)	23.62us (0.13us)
esp8266 during repair	26.75us (0.19us)	29.9us (0.15us)

Table 5: Average Put and Get performance with and without *SansSouci* in the cloud, at the edge, and on a device. Each average is over 1,000 back-to-back executions in a local namespace, the units are microseconds and the standard deviations are shown in parentheses.

In Table 5 we show the overheads that *SansSouci* adds to *CSPOT* both during normal operation (the common case) and during replay. Each benchmark runs 1,000 *CSPOT* operations (either Put or Get) back-to-back. The cloud and the device (microcontroller) are the same as that listed in Table 1. However, in this experiment we use a Raspberry Pi 3 Model B+ (RPi3) as the edge device. The RPi3 has 1 GB of RAM and a 1.4 GHz ARM Cortex CPU. The benchmark source is identical for all three deployments.

The results show that *SansSouci* adds little overhead, regardless of the hosting tier when integrated into *CSPOT*. This lack of overhead is especially important in the device tier where power considerations and battery life must be optimized. This efficiency illustrates the value of the *CSPOT* system architecture in that data and events are automatically tracked as part of normal runtime system operation (which is inherently efficient as illustrated in Table 2).

6.3 AWS Lambda Compatibility

To test the versatility of *CSPOT* as a low-level runtime system, we next implement a service that is interface-compatible with AWS Lambda and S3.The services allows users to execute AWS Lambda

	AWS Lambda rec/sec	*CSPOT*-Lambda rec/sec
1 Client	7.87	7.23
2 Clients	15.84	13.49
4 Clients	30.57	26.22

Table 6: Throughput (requests per second) for AWS Lambda and *CSPOT* Lambda when executing a Python Lambda function with 100 millisecond duration.

functions written in Python on *CSPOT*, without modification. The *CSPOT* Lambda and S3 services use only WooFs and handlers for their implementation. Thus, subject to the availability of a Python interpreter, this service makes AWS Lambda functions (invoked via the AWS Lambda API or via S3 updates) portable across all platforms (at all scales) on which *CSPOT* runs.

Table 6 shows a scaling comparison (in requests per second (req/sec)) between AWS Lambda and *CSPOT* Lambda when hosted in AWS. In this experiment, separate clients running in the same VM make concurrent Lambda function invocations using the AWS Lambda SDK/API for Python (each one executing for 100 milliseconds, which is the minimum charge duration for AWS Lambda) back-to-back. We run all experiments in the AWS us-west-1 region. We host *CSPOT* in in a c4.xlarge instance which has 4 virtual CPUs and 7.5 GB of RAM. Each datum is the average number of requests per second over 100 repeated invocations.

CSPOT Lambda and AWS Lambda compare favorably in that both achieve nearly linear scaling up to four clients. This result is somewhat startling because the *CSPOT* Lambda implementation uses only append-only WooFs (some with 16 KB payloads) to implement *all* of its data structures (including those needed to implement the AWS Lambda and S3 APIs). The *CSPOT* service consists of two separate Linux processes (both written in a mix of C and C++) that make native *CSPOT* API calls through the C interface for *CSPOT*. To track objects within S3 buckets requires accessing several internal data structures that are mapped to WooFs for persistent storage. Clearly this implementation cannot achieve the scaling of AWS Lambda. However, at small scales (i.e. scales that occur at the device and edge tiers) *CSPOT* is comparable *even when run in the AWS public cloud.*

6.4 Virtualization Performance Impact

Our *CSPOT* implementation is sufficiently new and different to expose previously undocumented performance interactions between the various virtualization technologies that it comprises. Specifically, while benchmarking *CSPOT* for this study, we noticed an unexpected performance characteristic for *CSPOT* when executed on a cloud. To illuminate it, we repeat the latency probe we use to generate the data in Table 2 using successively larger "put" sizes. We do so as a way of measuring the throughput *CSPOT* can sustain.

Figure 4 shows the maximum throughput (y-axis) associated with successively larger put-payload sizes (x-axis) for *CSPOT* running on the edge cloud (blue), campus cloud (amber), and in AWS (green). In addition, we deployed *CSPOT* on the edge cloud natively without virtualization (red).

This data indicates that operating system virtualization cuts maximum throughput by a factor of two (the results are similar for

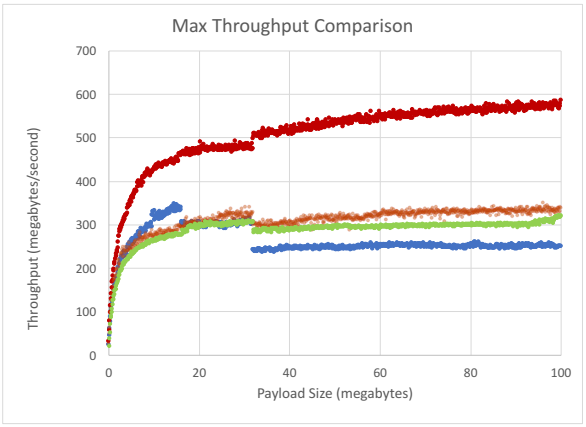

Figure 4: Comparison of maximum throughput rates as a function of payload size across edge and cloud platforms. Red features are for unvirtualized edge (NUC) execution, blue is virtualized NUC (edge cloud (NUC-VM)), amber is for campus private cloud, and green is AWS.

average throughput but are omitted for brevity). Also, these maximum throughput graphs show smoothly increasing throughput as a function of payload size up to some host-specific maximum. Curiously, *all* of the virtualized deployments show a sharp drop in maximum throughput, with the unvirtualized deployment showing an increase, when the payload size exceeds 32M bytes.

Recall that *CSPOT* runs all handlers as processes within a Docker (v18.09) container associated with a namespace on Linux hosts. In clouds, running Docker in a VM is commonplace to implement Linux namespace [58] and cgroup [57] isolation between applications using the VM. However, because *CSPOT* applications are limited to FaaS computational semantics and WooF storage, this double layer of isolation (Linux container and VM) may not be necessary, especially if it carries a substantial performance penalty. Docker is heavily used in clouds but despite its popularity, we are unaware of literature documenting this level of performance degradation. Thus, it may be that *CSPOT*'s use of mapped virtual memory to implement fast append-only semantics is exposing a rare interplay between Docker container and VM isolation mechanisms. Clearly, *CSPOT* is "stressing" well-used and mature virtualization technologies in new ways. However, because of this obvious difference, as part of our future work, we plan to study whether *CSPOT* is sufficient to serve as its own virtualization technology for edge devices or whether it requires a hypervisor for additional isolation and configuration support.

Taken together, these results indicate that *CSPOT* achieves very low latencies compared to AWS Lambda (cf Table 2) but the combination of hypervisor virtualization and Linux containers (used by Docker) has a dramatic effect on throughput. Specifically, virtualized deployment of *CSPOT* appears to experience significantly greater variability in throughput compared to an unvirtualized deployment (with similar tenancy competition). Moreover, the maximum throughput appears almost halved when *CSPOT* is deployed in *any* cloud setting relative to an unvirtualized deployment on a

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)

Table 7: Comparison of cryptographic algorithms for signing and verifying a 32 byte message on the esp8266. Average execution time and standard deviation (in parenthesis) are shown. The last 2 rows show the performance when *CSPOT* performs two (3-level Derived) and four (5-level Derived) attenuations on the capability.

small, portable system (i.e. the Intel NUC used in this study). These results indicate that, perhaps, the currently-popular cloud isolation technologies are imposing a significant latency penalty when the runtime system is lightweight – a result that warrants further study.

6.5 Security Performance

To illustrate the performance impact of *CSPOT* 's capability-based security mechanisms, we next benchmark different signing mechanisms when a *CSPOT* server is running on an esp8266 microcontroller using its built-in WiFi wireless network. We show signing times (needed for capability generation and attenuation) and verification times on the microcontroller. In each experiment the microcontroller is communicating with an Intel NUC attached to the same wireless network which is acting as an edge device.

Table 7 shows the comparison of the execution times in milliseconds for various cryptographic techniques when used to sign (column 2) and verify (column 3) messages on the microcontroller. 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 [43]), and the HMAC-based scheme that we employ in *CSPOT* for two key lengths. Below the double lines we also show the performance of a 3-level capability derivation (2 attenuations) and a 5-level derivation (4 attenuations) on the device, which represent the setting in which clients attenuate their capabilities multiple times.

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

6.6 Reliability

Finally, our experience with *CSPOT* is that it has proven to be remarkably reliable. At the time of this writing, we have been using *CSPOT* as the data acquisition and stream processing infrastructure for two long-lived IoT deployments in agricultural settings. The first monitors soil moisture readings in an almond orchard located in Fresno, California and the second is the basis for the frost prediction and prevention system (described in Section 5) located in a citrus orchard in Exeter, California. *CSPOT* is running unattended in these locations, both virtualized and unvirtualized, as well as on multiple edge devices and campus private cloud. These production *CSPOT* deployments have been operational since August 1, 2017 (although the deployment has grown steadily since then) and has fired approximately 6.8 million handlers (averaging one every 6 seconds) during this period without a detected software failure. By way of comparison, the campus networking infrastructure (staffed 24/7) experienced one major and two minor outages during the same period.

This reliability is surprising, given the novel interaction between containers, memory mapped files, threads, and Linux processes that the *CSPOT* runtime requires. Indeed, especially given the speed with which *CSPOT* is able to invoke a handler within a container, our expectation is that it would expose synchronization or file-system reliability issues in Linux, which it has not to date.

7 RELATED WORK

The utility of serverless computing for different workloads has been studied recently for various application types [17, 33, 41, 50, 51, 69, 85]. In [41], the authors identify key weaknesses in existing serverless implementations which we believe we are the first to attempt to overcome. These weaknesses are the overhead imposed by the requirement that FaaS functions communicate via slow (versus point to point networking), remote, location-concealed services. *CSPOT* introduces persistent, append-only (i.e. versioned) storage abstractions and namespace to the serverless model to significantly improve performance and to facilitate co-location of related code and data. At the same time, these abstractions enable storage to be geo-distributed and replicated for durability and robustness, and dependent events to be tracked for use as a concurrent programming aid. The authors of [62] investigate adding retroactive programming (i.e. support for reprogramming application histories via event and state-update tracking) to serverless. *CSPOT* provides a runtime system over which retroactive programming can be implemented (by combining its persistent, append-only data structures data dependencies, and event logs).

From a technological perspective, the most closely related work to *CSPOT* is AWS Greengrass [2] and Microsoft Azure IoT Edge [65] over IoT Hub [66]. These commercial systems are relatively mature offerings designed to allow edge systems to act as intermediaries between devices and their respective public cloud services. *CSPOT* differs from these systems in that it couples FaaS function invocation with persistent storage wherever it runs – on a device, at the edge, in a private cloud, or in a public cloud. *CSPOT* microcontroller support also differs in that uses the same programming model and runtime system across IoT tiers. Azure and AWS require a combination of potentially conflicting and confusing technologies to be integrated, deployed, and managed by applications developers (MQTT, FaaS services, and disparate libraries, authentication mechanisms, and configuration). Finally, *CSPOT* is open source and works across clouds.

CSPOT is designed for a multi-tiered "device-edge-cloud" architecture like that described in [29] and [67]. Like the authors of these works, we hypothesize that IoT will require a hierarchical set of computational, storage, and network resources. Uniquely, however,

CSPOT defines portable, high-performance FaaS abstractions and enables code and data mobility without modification throughout the complete hierarchy.

In that vein, *CSPOT* shares conceptual firmament with OpenWhisk [18], OpenFaaS [75] and OpenLambda [42]. These systems are open source and implement the FaaS paradigm using containers for isolation. In addition, OpenFaaS uses Linux processes and binaries as the execution mechanism within a container, as does *CSPOT*. These efforts are distinct however, in that they are not designed for device or edge computing in general, and IoT in particular. Further, *CSPOT* defines specific low-level abstractions that are intended to support a variety of higher level language and distributed systems technologies. These other technologies are designed to implement FaaS as a language-level platform.

In [26] and [71] the authors review many of the full-stack challenges that must be overcome to realize "The Internet of Things" as a society-serving technology. While these top-down approaches clearly frame the issues and offer possible avenues of exploration, in contrast, our work is best classified as "bottom up." *CSPOT* starts by defining a set of low-level abstractions that can then be composed to create higher-level functionality. While eliding a complete architecture for "The Internet of Things," we validate *CSPOT* functionality with working IoT applications that serve non-expert users.

Because of its bottom up approach, *CSPOT* is similar to [5], which constructs a logical architecture for IoT applications as a composition of individually developed technologies and services. It provides reference implementations for these services and validates its results empirically. *CSPOT* is far less broad in its scope and could easily serve as a single component within the system described in [5]. It is, in the same way, portable, high performance, and empirically validated while, at the same time, leveraging cloud technologies (e.g. FaaS) at a variety of execution scales.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we present *CSPOT*, a FaaS platform for IoT application development and deployment. *CSPOT* provides a low-level implementation and APIs that facilitate application portability across IoT tiers, tracking of causal dependencies across a distributed application, extensibility for easy integration of new IoT services, and low-latency response for IoT applications. We detail *CSPOT*'s abstractions as well as its novel programming, memory, and persistence models that together enable these features.

We also provide examples for the *CSPOT* APIs and applications, and empirically evaluate it using heterogeneous systems and multiple performance metrics. In addition, we empirically evaluate novel services (i) for persistent data repair and function replay, (ii) for AWS Lambda and S3 Python compatibility, and (iii) for capability-based security. Our results show that, relative to extant FaaS systems, *CSPOT* improves the response time and facilitates low-overhead end-to-end performance for IoT benchmarks and applications across IoT tiers/scales.

As part of future work, we are extending *CSPOT* to other popular microcontroller systems, supporting functions in high-level languages, and investigating novel runtime extensions that facilitate large scale deployment debugging, and that optimize performance, energy use, and cost (of public cloud use) across tiers. We are also investigating support for scalable service/device discovery and virtualization, privacy preservation, and robust and transparent disconnected operation.

REFERENCES

[1] P. Alvaro, S. Galwani, and P. Bailis. 2017. Research for Practice: Tracing and Debugging Distributed Systems; Programming by Examples. In *CACM*.

[2] Amazon. 2017. Amazon GreenGrass. "https://aws.amazon.com/greengrass/" Accessed 15-Sep-2017.

[3] Amazon DynamoDB 2016. Amazon DynamoDB. https://aws.amazon.com/dynamodb/. [Online; accessed 15-Nov-2016].

[4] Amazon IoT SDK 2018. "https://docs.aws.amazon.com/iot/latest/developerguide/iot-sdks.html" Accessed 7-May-2018.

[5] M. Andersen, G. Fierro, and D. Culler. 2017. Enabling synergy in iot: Platform to service and beyond. *Network and Computer Applications* 81 (2017).

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

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

[8] AWS Lambda for Microservices 2019. AWS Lambda for Microservices. "https://aws.amazon.com/microservices/" [Online; accessed on 12-Aug-2019].

[9] AWS Lambda for Web Services 2019. AWS Lambda for Web Services. "https://aws.amazon.com/getting-started/projects/build-serverless-web-app-lambda-apigateway-s3-dynamodb-cognito/" [Online; accessed on 12-Aug-2019].

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

[11] AWS Simple Storage Service (S3) 2019. AWS Simple Storage Service (S3). "http://aws.amazon.com/s3/".

[12] AWS X-Ray 2017. https://aws.amazon.com/xray/. [Online; accessed 11-Sep-2017].

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

[14] F. Bakir, R. Wolski, C. Krintz, and G. Sankar Ramachandran. 2019. Devices-as-Services: Rethinking Scalable Service Architectures for the Internet of Things. In *USENIX HotEdge*.

[15] Mahesh Balakrishnan, Dahlia Malkhi, Vijayan Prabhakaran, Ted Wobbler, Michael Wei, and John D Davis. 2012. Corfu: A shared log design for flash clusters. In *USENIX NSDI*.

[16] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, et al. 2017. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*. Springer, 1–20.

[17] I. Baldini, P. Castro, K. Chang, P. Cheng, S. Fink, V. Ishakian, N. Mitchell, V. Muthusamy, R. M. Rabbah, A. Slominski, and P. Suter. 2017. Serverless Computing: Current Trends and Open Problems. *CoRR* (2017).

[18] Ioana Baldini, Paul Castro, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, and Philippe Suter. 2016. Cloud-native, event-based programming for mobile applications. In *Mobile Software Engineering and Systems*.

[19] Richard Barry. 2009. *FreeRTOS reference manual: API functions and configuration options.* Real Time Engineers Limited.

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

[21] I. Beschastnikh, Y. Brun, M. D. Ernst, A. Krishnamurthy, and T. E. Anderson. 2012. Mining Temporal Invariants from Partially Ordered Logs. *SIGOPS Oper. Syst. Rev.* 45, 3 (Jan. 2012).

[22] I. Beschastnikh, P. Wang, Y. Brun, and M. Ernst. 2016. Debugging distributed systems. In *CACM*.

[23] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. 2014. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. In *Network and Distributed System Security Symposium*.

[24] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. 2012. Fog computing and its role in the internet of things. In *Mobile cloud computing workshop*.

[25] Eric A. Brewer. 2000. Towards Robust Distributed Systems (Abstract). In *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*.

[26] Mung Chiang and Tao Zhang. 2016. Fog and IoT: An overview of research opportunities. *IEEE Internet of Things Journal* 3, 6 (2016), 854–864.

[27] Docker 2016. Docker. https://www.docker.com [Online; accessed 1-Nov-2016].

[28] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. 1989. Making Data Structures Persistent. *J. Comput. Syst. Sci.* 38, 1 (Feb. 1989).

[29] Andy Rosales Elias, Nevena Golubovic, Chandra Krintz, and Rich Wolski. 2017. Where's The Bear?–Automating Wildlife Image Processing Using IoT and Edge Cloud Systems. In *ACM Conference on IoT Design and Implementation*.

[30] Espressif 8266 Microcontroller 2018. Espressif 8266 Microcontroller. https://en.wikipedia.org/wiki/ESP8266. [Accessed electronically, September 2018].

[31] Eucalyptus Open Source Project. 2018. http://www.eucalyptus.cloud.

[32] D. Floyer. 2019. The Vital Role of Edge Computing inthe Internet of Things. http://wikibon.com/the-vital-role-of-edge-computing-in-the-internet-of-things/ [Online; accessed 22-Aug-2019].

[33] S. Fouladi, R. Wahby, B. Shacklett, K. Balasubramaniam, W. Zeng, R. Bhalerao, A. Sivaraman, G. Porter, and K. Winstein. 2017. Encoding, Fast and Slow: Low-Latency Video Pro- cessing Using Thousands of Tiny Threads. In *NSDI*.

[34] Apache Foundation. 2017. *HDFS*. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[35] Elijah Gabriel. 2019. Cloudlet-based Mobile Computing. "http://elijah.cs.cmu.edu/" [Online; accessed 22-Aug-2019].

[36] D. Geels, G. Altekar, P. Maniatis, T. Roscoe, and I. Stoica. 2007. Friday: Global Comprehension for Distributed Replay. In *NSDI*.

[37] N. Golubovic, R. Wolski, C. Krintz, and M. Mock. 2019. Improving the Accuracy of Outdoor Temperature Prediction by IoT Devices. In *IEEE Conference on IoT*.

[38] Nina Golyandina and Anatoly Zhigljavsky. 2013. *Singular Spectrum Analysis for time series*. Springer Science & Business Media.

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

[40] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2018. Serverless Computing: One Step Forward, Two Steps Back. *arXiv preprint arXiv:1812.03651* (2018).

[41] Joseph M. Hellerstein, Jose Faleiro, Joseph E. Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. 2019. Serverless Computing: One Step Forward, Two Steps Back. In *Conference on Innovative Data Systems Research (CIDR)*.

[42] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. 2016. Serverless Computation with openLambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing (HotCloud'16)*.

[43] J Hernandez-Ramos, A. Jara, L. Marin, and A. Skarmeta Gomez. 2016. DCapBAC: Embedding Authorization Logic into Smart Things Through ECC Optimizations. *Int. J. Comput. Math.* 93, 2 (Feb. 2016).

[44] Urs Hunkeler, Hong Linh Truong, and Andy Stanford-Clark. 2008. MQTT-S-A publish/subscribe protocol for Wireless Sensor Networks. In *Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on*. IEEE, 791–798.

[45] IBM OpenWhisk 2016. IBM OpenWhisk. https://developer.ibm.com/openwhisk/. [Online; accessed 15-Nov-2016].

[46] Intel NUC 6i7KYK 2018. https://www.intel.com/content/www/us/en/nuc/nuc-kit-nuc6i7kyk-features-configurations.html [Online; accessed April 2018].

[47] Iron.io 2016. Iron.io. https://www.iron.io. [Online; accessed 15-Nov-2016].

[48] Hiranya Jayathilaka, Chandra Krintz, and Rich Wolski. 2017. Performance Monitoring and Root Cause Analysis for Cloud-hosted Web Applications. In *Proceedings of WWW 2017 (to appear)*.

[49] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the cloud: Distributed computing for the 99%. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 445–451.

[50] Eric Jonas, Qifan Pu, Shivaram Venkataraman, Ion Stoica, and Benjamin Recht. 2017. Occupy the Cloud: Distributed Computing for the 99%. In *SOCC*.

[51] A. Klimovic, Y. Wang, C. Kozyrakis, P. Stuedi, J. Pfefferle, and A. Trivedi. 2018. Understanding Ephemeral Storage for Serverless Analytics. In *USENIX ATC*.

[52] Ramakrishna Kotla, Lorenzo Alvisi, and Mike Dahlin. 2007. SafeStore: A durable and practical storage system. In *USENIX Annual Technical Conference*. 129–142.

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

[54] T. Li, K. Keahey, K. Wang, D. Zhao, and I. Raicu. 2015. A dynamically scalable cloud data infrastructure for sensor networks. In *Workshop on Scientific Cloud Computing*.

[55] W-T. Lin, F. Bakir, C. Krintz, R. Wolski, and M. Mock. 2019. Data repair for Distributed, Event-based IoT Applications. In *ACM International Conference on Distributed and Event-Based Systems*.

[56] W-T. Lin, C. Krintz, R. Wolski, and M. Zhang. 2017. Tracking Causal Order in AWS Lambda Applications. In *IEEE International Conference on Cloud Engineering*.

[57] Linux cgroups 2018. Linux Cgroups Man Page. http://man7.org/linux/man-pages/man7/cgroups.7.html. [Accessed electronically, September 2018].

[58] Linux namespace 2018. Linux Namespace Man Page. http://man7.org/linux/man-pages/man7/namespaces.7.html. [Accessed electronically, September 2018].

[59] Linux Semaphores 2018. Linux Semaphores. http://man7.org/linux/man-pages/man7/sem_overview.7.html. [Accessed electronically, March 2018].

[60] Xuezheng Liu, Wei Lin, Aimin Pan, and Zheng Zhang. 2007. WiDS Checker: Combating Bugs in Distributed Systems. In *NSDI*.

[61] J. Mace, R. Roelke, and R. Fonseca. 2015. Pivot Tracing: Dynamic Causal Monitoring for Distributed Systems. In *Symposium on Operating System Principles*.

[62] Dominik Meissner, Benjamin Erb, Frank Kargl, and Matthias Tichy. 2018. Retro-Lambda: An Event-sourced Platform for Serverless Applications with Retroactive Computing Support. In *Intl. Conf. on Distributed and Event-based Systems*.

[63] Message Broker Protocols for AWS 2018. Message Broker Protocols for AWS. https://docs.aws.amazon.com/iot/latest/developerguide/protocols.html. [Accessed electronically, September 2018].

[64] Microsoft. 2018. Microsoft Azure CosmosDB. "https://azure.microsoft.com/en-us/services/cosmos-db/" Accessed May-2018.

[65] Microsoft. 2018. Microsoft Azure IoT Edge. "https://azure.microsoft.com/en-us/services/iot-edge/" Accessed May 2018.

[66] Microsoft. 2018. Microsoft Azure IoT Hub. "https://azure.microsoft.com/en-us/services/iot-hub/" Accessed May 2018.

[67] S. Mortazavi, M. Salehe, C. Gomes, C. Phillips, and E. de Lara. 2017. Cloudpath: a multi-tier cloud computing framework. In *ACM/IEEE Symposium on Edge Computing*.

[68] msgpack 2018. MessagePack Protocol. https://msgpack.org/index.html. [Accessed electronically, September 2018].

[69] N. Mukhi, S. Prabhu, and B. Slawson. 2017. Using a Serverless Framework for Implementing a Cognitive Tutor: Experiences and Issues. In *Intl Workshop on Serverless Computing*.

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

[71] Arslan Munir, Prasanna Kansakar, and Samee U Khan. 2017. IFCIoT: Integrated Fog Cloud IoT: A novel architectural paradigm for the future Internet of Things. *IEEE Consumer Electronics Magazine* 6, 3 (2017), 74–82.

[72] James Murty. 2009. *Programming Amazon Web Services: S3, EC2, SQS, FPS, and SimpleDB*. O'Reilly Media, Inc.

[73] Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and Amr El Abbadi. 2015. Chariots: A Scalable Shared Log for Data Management in Multi-Datacenter Cloud Environments.. In *EDBT*. 13–24.

[74] Daniel Nurmi, Richard Wolski, Chris Grzegorczyk, Graziano Obertelli, Sunil Soman, Lamia Youseff, and Dmitrii Zagorodnov. 2009. The eucalyptus open-source cloud-computing system. In *Cluster Computing and the Grid, 2009. CCGRID'09. 9th IEEE/ACM International Symposium on*. IEEE, 124–131.

[75] OpenFaaS 2018. https://www.openfaas.com. [Accessed electronically, May 2018].

[76] Raspberry Pi Zero 2018. Raspberry Pi Zero. https://www.raspberrypi.org/products/raspberry-pi-zero/. [Accessed electronically, March 2018].

[77] Raspian OS 2018. Raspian OS. https://www.raspberrypi.org/downloads/raspbian/. [Accessed electronically, March 2018].

[78] E. Rescorla. 2018. *The Transport Layer Security (TLS) Protocol Version 1.3*. RFC 8446. IETF.

[79] Mendel Rosenblum and John K Ousterhout. 1992. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems (TOCS)* 10, 1 (1992), 26–52.

[80] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. 2010. The Hadoop Distributed File System. In *IEEE Symposium on Mass Storage Systems and Technologies*.

[81] S. Simanta, K. Ha, G. Lewis, E. Morris, and M. Satyanarayanan. 2012. A Reference Architecture for Mobile Code Offload in Hostile Environments. In *Intl. Conf. on Mobile Computing, Applications and Services*. http://elijah.cs.cmu.edu/DOCS/cloudlet_hostile_MobiCASE2012_camera_ready.pdf [Online; accessed 22-Apr-2019].

[82] Y. Simmhan, C. Van Ingen, G. Subramanian, and J. Li. 2010. Bridging the gap between desktop and the cloud for escience applications. In *IEEE Cloud*.

[83] Stephen Soltesz, Herbert Pötzl, Marc E Fiuczynski, Andy Bavier, and Larry Peterson. 2007. Container-based operating system virtualization: a scalable, high-performance alternative to hypervisors. *ACM SIGOPS Operating Systems Review* 41, 3 (2007), 275–287.

[84] The Linux mmap system call 2018. The Linux mmap system call. http://man7.org/linux/man-pages/man2/mmap.2.html. [Accessed electronically, March 2018].

[85] A. Tumanov, T. Zhu, J. Park, M. Kozuch, M. Harchol-Balter, and G. Ganger. 2016. TetriSched: global rescheduling with adaptive plan-ahead in dynamic heterogeneous clusters. In *EuroSys*.

[86] Robbert Van Renesse and Fred B Schneider. 2004. Chain Replication for Supporting High Throughput and Availability.. In *OSDI*, Vol. 4.

[87] Tim Verbelen, Pieter Simoens, Filip De Turck, and Bart Dhoedt. 2012. Cloudlets: bringing the cloud to the mobile user. In *ACM workshop on Mobile cloud computing and services*. ACM.

[88] Hoang Tam Vo, Sheng Wang, Divyakant Agrawal, Gang Chen, and Beng Chin Ooi. 2012. LogBase: a scalable log-structured database system in the cloud. *VLDB* 5, 10 (2012).

[89] Zeromq Desitributed Messaging 2018. Zeromq Desitributed Messaging. http://zeromq.org. [Accessed electronically, March 2018].