Firecracker: Lightweight Virtualization for Serverless Applications

Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache,
Anthony Liguori, Rolf Neugebauer, Phil Piwonka,
and Diana-Maria Popa, *Amazon Web Services*

https://www.usenix.org/conference/nsdi20/presentation/agache

This paper is included in the Proceedings of the
17th USENIX Symposium on Networked Systems Design
and Implementation (NSDI '20)
February 25–27, 2020 • Santa Clara, CA, USA

978-1-939133-13-7

Firecracker: Lightweight Virtualization for Serverless Applications

Alexandru Agache
Amazon Web Services

Marc Brooker
Amazon Web Services

Andreea Florescu
Amazon Web Services

Alexandra Iordache
Amazon Web Services

Anthony Liguori
Amazon Web Services

Rolf Neugebauer
Amazon Web Services

Phil Piwonka
Amazon Web Services

Diana-Maria Popa
Amazon Web Services

Abstract

Serverless containers and functions are widely used for deploying and managing software in the cloud. Their popularity is due to reduced cost of operations, improved utilization of hardware, and faster scaling than traditional deployment methods. The economics and scale of serverless applications demand that workloads from multiple customers run on the same hardware with minimal overhead, while preserving strong security and performance isolation. The traditional view is that there is a choice between virtualization with strong security and high overhead, and container technologies with weaker security and minimal overhead. This tradeoff is unacceptable to public infrastructure providers, who need both strong security and minimal overhead. To meet this need, we developed Firecracker, a new open source Virtual Machine Monitor (VMM) specialized for serverless workloads, but generally useful for containers, functions and other compute workloads within a reasonable set of constraints. We have deployed Firecracker in two publically-available serverless compute services at Amazon Web Services (Lambda and Fargate), where it supports millions of production workloads, and trillions of requests per month. We describe how specializing for serverless informed the design of Firecracker, and what we learned from seamlessly migrating Lambda customers to Firecracker.

1 Introduction

Serverless computing is an increasingly popular model for deploying and managing software and services, both in public cloud environments, e.g., [4, 16, 50, 51], as well as in on-premises environments, e.g., [11, 41]. The serverless model is attractive for several reasons, including reduced work in operating servers and managing capacity, automatic scaling, pay-for-use pricing, and integrations with sources of events and streaming data. Containers, most commonly embodied by Docker, have become popular for similar reasons, including reduced operational overhead, and improved manageability. Containers and Serverless offer a distinct economic advantage over traditional server provisioning processes: multitenancy allows servers to be shared across a large number of workloads, and the ability to provision new functions and containers in milliseconds allows capacity to be switched between workloads quickly as demand changes. Serverless is also attracting the attention of the research community [21,26,27,44,47], including work on scaling out video encoding [13], linear algebra [20, 53] and parallel compilation [12].

Multitenancy, despite its economic opportunities, presents significant challenges in isolating workloads from one another. Workloads must be isolated both for security (so one workload cannot access, or infer, data belonging to another workload), and for operational concerns (so the noisy neighbor effect of one workload cannot cause other workloads to run more slowly). Cloud instance providers (such as AWS EC2) face similar challenges, and have solved them using hypervisor-based virtualization (such as with QEMU/KVM [7, 29] or Xen [5]), or by avoiding multi-tenancy and offering bare-metal instances. Serverless and container models allow many more workloads to be run on a single machine than traditional instance models, which amplifies the economic advantages of multi-tenancy, but also multiplies any overhead required for isolation.

Typical container deployments on Linux, such as those using Docker and LXC, address this density challenge by relying on isolation mechanisms built into the Linux kernel. These mechanisms include *control groups* (*cgroups*), which provide process grouping, resource throttling and accounting; *namespaces*, which separate Linux kernel resources such as process IDs (PIDs) into namespaces; and *seccomp-bpf*, which controls access to syscalls. Together, these tools provide a powerful toolkit for isolating containers, but their reliance on a single operating system kernel means that there is a fundamental tradeoff between security and code compatibility. Container implementors can choose to improve security by limiting syscalls, at the cost of breaking code which requires the restricted calls. This introduces difficult tradeoffs: implementors of serverless and container services can choose between hypervisor-based virtualization (and the potentially unacceptable overhead related to it), and Linux containers (and the related compatibility vs. security tradeoffs). We built Firecracker because we didn't want to choose.

Other projects, such as Kata Containers [14], Intel's Clear Containers, and NEC's LightVM [38] have started from a similar place, recognizing the need for improved isolation, and choosing hypervisor-based virtualization as the way to achieve that. QEMU/KVM has been the base for the majority of these projects (such as Kata Containers), but others (such as LightVM) have been based on slimming down Xen. While QEMU has been a successful base for these projects, it is a large project ($>$ 1.4 million LOC as of QEMU 4.2), and has focused on flexibility and feature completeness rather than overhead, security, or fast startup.

With Firecracker, we chose to keep KVM, but entirely replace QEMU to build a new Virtual Machine Monitor (VMM), device model, and API for managing and configuring MicroVMs. Firecracker, along with KVM, provides a new foundation for implementing isolation between containers and functions. With the provided minimal Linux guest kernel configuration, it offers memory overhead of less than 5MB per container, boots to application code in less than 125ms, and allows creation of up to 150 MicroVMs per second per host. We released Firecracker as open source software in December 2018[1], under the Apache 2 license. Firecracker has been used in production in Lambda since 2018, where it powers millions of workloads and trillions of requests per month.

Section 2 explores the choice of an isolation solution for Lambda and Fargate, comparing containers, language VM isolation, and virtualization. Section 3 presents the design of Firecracker. Section 4 places it in context in Lambda, explaining how it is integrated, and the role it plays in the performance and economics of that service. Section 5 compares Firecracker to alternative technologies on performance, density and overhead.

1.1 Specialization

Firecracker was built specifically for serverless and container applications. While it is broadly useful, and we are excited to see Firecracker be adopted in other areas, the performance, density, and isolation goals of Firecracker were set by its intended use for serverless and containers. Developing a VMM for a clear set of goals, and where we could make assumptions about the properties and requirements of guests, was significantly easier than developing one suitable for all uses. These simplifying assumptions are reflected in Firecracker's design and implementation. This paper describes Firecracker in context, as used in AWS Lambda, to illustrate why we made the decisions we did, and where we diverged from existing VMM designs. The specifics of how Firecracker is used in Lambda are covered in Section 4.1.

Firecracker is probably most notable for what it does not offer, especially compared to QEMU. It does not offer a BIOS, cannot boot arbitrary kernels, does not emulate legacy devices nor PCI, and does not support VM migration. Firecracker could not boot Microsoft Windows without significant changes to Firecracker. Firecracker's process-per-VM model also means that it doesn't offer VM orchestration, packaging, management or other features — it replaces QEMU, rather than Docker or Kubernetes, in the container stack. Simplicity and minimalism were explicit goals in our development process. Higher-level features like orchestration and metadata management are provided by existing open source solutions like Kubernetes, Docker and `containerd`, or by our proprietary implementations inside AWS services. Lower-level features, such as additional devices (USB, PCI, sound, video, etc), BIOS, and CPU instruction emulation are simply not implemented because they are not needed by typical serverless container and function workloads.

2 Choosing an Isolation Solution

When we first built AWS Lambda, we chose to use Linux containers to isolate functions, and virtualization to isolate between customer accounts. In other words, multiple functions for the same customer would run inside a single VM, but workloads for different customers always run in different VMs. We were unsatisfied with this approach for several reasons, including the necessity of trading off between security and compatibility that containers represent, and the difficulties of efficiently packing workloads onto fixed-size VMs. When choosing a replacement, we were looking for something that provided strong security against a broad range of attacks (including microarchitectural side-channel attacks), the ability to run at high densities with little overhead or waste, and compatibility with a broad range of unmodified software (Lambda functions are allowed to contain arbitrary Linux binaries, and a significant portion do). In response to these challenges, we evaluated various options for re-designing Lambda's isolation model, identifying the properties of our ideal solution:

Isolation: It must be safe for multiple functions to run on the same hardware, protected against privilege escalation, information disclosure, covert channels, and other risks.

Overhead and Density: It must be possible to run thousands of functions on a single machine, with minimal waste.

Performance: Functions must perform similarly to running natively. Performance must also be consistent, and isolated from the behavior of neighbors on the same hardware.

Compatibility: Lambda allows functions to contain arbitrary Linux binaries and libraries. These must be supported without code changes or recompilation.

[1] https://firecracker-microvm.github.io/

Fast Switching: It must be possible to start new functions and clean up old functions quickly.

Soft Allocation: It must be possible to over commit CPU, memory and other resources, with each function consuming only the resources it needs, not the resources it is entitled to.

Some of these qualities can be converted into quantitative goals, while others (like isolation) remain stubbornly qualitative. Modern commodity servers contain up to 1TB of RAM, while Lambda functions use as little as 128MB, requiring up to 8000 functions on a server to fill the RAM (or more due to soft allocation). We think of overhead as a percentage, based on the size of the function, and initially targeted 10% on RAM and CPU. For a 1024MB function, this means 102MB of memory overhead. Performance is somewhat complex, as it is measured against the function's entitlement. In Lambda, CPU, network, and storage throughput is allocated to functions proportionally to their configured memory limit. Within these limits, functions should perform similarly to bare metal on raw CPU, IO throughput, IO latency and other metrics.

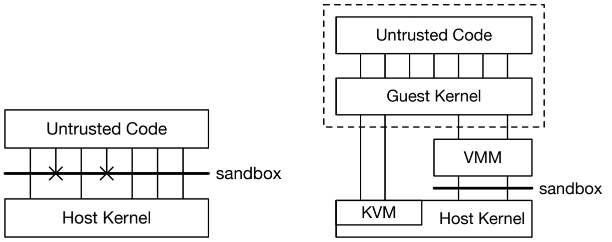
2.1 Evaluating the Isolation Options

Broadly, the options for isolating workloads on Linux can be broken into three categories: *containers*, in which all workloads share a kernel and some combination of kernel mechanisms are used to isolate them; *virtualization*, in which workloads run in their own VMs under a hypervisor; and *language VM isolation*, in which the language VM is responsible for either isolating workloads from each other or from the operating system. [2]

Figure 1 compares the security approaches between Linux containers and virtualization. In Linux containers, untrusted code calls the host kernel directly, possibly with the kernel surface area restricted (such as with seccomp-bpf). It also interacts directly with other services provided by the host kernel, like filesystems and the page cache. In virtualization, untrusted code is generally allowed full access to a guest kernel, allowing all kernel features to be used, but explicitly treating the guest kernel as untrusted. Hardware virtualization and the VMM limit the guest kernel's access to the privileged domain and host kernel.

2.1.1 Linux Containers

Containers on Linux combine multiple Linux kernel features to offer operational and security isolation. These features include: *cgroups*, providing CPU, memory and other resource limits; *namespaces*, providing namespacing for kernel resources like user IDs (uids), process IDs (pids) and network interfaces; *seccomp-bpf*, providing the ability to limit which syscalls a process can use, and which arguments can be passed to these syscalls; and *chroot*, proving an isolated filesystem. Different Linux container implementations use these tools in different combinations, but *seccomp-bpf* provides the most important security isolation boundary. The fact that containers rely on syscall limitations for their security represents a tradeoff between security and compatibility. A trivial Linux application requires 15 unique syscalls. Tsai et al [57] found that a typical Ubuntu Linux 15.04 installation requires 224 syscalls and 52 unique *ioctl* calls to run without problems, along with the */proc* and */sys* interfaces of the kernel. Nevertheless, meaningful reduction of the kernel surface is possible, especially as it is reasonable to believe that the Linux kernel has more bugs in syscalls which are less heavily used [32].



(a) Linux container model (b) KVM virtualization model

Figure 1: The security model of Linux containers (a) depends directly on the kernel's sandboxing capabilities, while KVM-style virtualization (b) relies on security of the VMM, possibly augmented sandboxing

One approach to this challenge is to provide some of the operating system functionality in userspace, requiring a significantly smaller amount of kernel functionality to provide the programmer with the appearance of a fully featured environment. Graphene [56], Drawbridge [45], Bascule [6] and Google's gvisor [15] take this approach. In environments running untrusted code, container isolation is not only concerned with preventing privilege escalation, but also in preventing information disclosure side channels (such as [19]), and preventing communication between functions over covert channels. The richness of interfaces like */proc* have showed this to be challenging (CVE-2018-17972 and CVE-2017-18344 are recent examples).

2.1.2 Language-Specific Isolation

A second widely-used method for isolating workloads is leveraging features of the language virtual machine, such as the Java Virtual Machine (JVM) or V8. Some language VMs (such as V8's *isolates* and the JVM's security managers) aim to run multiple workloads within a single process, an approach which introduces significant tradeoffs between functionality

[2] It's somewhat confusing that in common usage *containers* is both used to describe the mechanism for packaging code, and the typical implementation of that mechanism. Containers (the abstraction) can be provided without depending on containers (the implementation). In this paper, we use the term *Linux containers* to describe the implementation, while being aware that other operating systems provide similar functionality.

(and compatibility) and resistance to side channel attacks such as Spectre [30, 39]. Other approaches, such as Chromium site isolation [46], Alto [31] and SAND [1] use a process per trust domain, and instead aim to prevent the code from escaping from the process or accessing information from beyond the process boundary. Language-specific isolation techniques were not suitable for Lambda or Fargate, given our need to support arbitrary binaries.

2.1.3 Virtualization

Modern virtualization uses hardware features (such as Intel VT-x) to provide each sandbox an isolated environment with its own virtual hardware, page tables, and operating system kernel. Two key, and related, challenges of virtualization are density and overhead. The VMM and kernel associated with each guest consumes some amount of CPU and memory before it does useful work, limiting density. Another challenge is startup time, with typical VM startup times in the range of seconds. These challenges are particularly important in the Lambda environment, where functions are small (so relative overhead is larger), and workloads are constantly changing. One way to address startup time is to boot something smaller than a full OS kernel, such as a unikernel. Unikernels are already being investigated for use with containers, for example in LightVM [38] and Solo5 [59]. Our requirement for running unmodified code targeting Linux meant that we could not apply this approach.

The third challenge in virtualization is the implementation itself: hypervisors and virtual machine monitors (VMMs), and therefore the required *trusted computing base* (TCB), can be large and complex, with a significant attack surface. This complexity comes from the fact that VMMs still need to either provide some OS functionality themselves (type 1) or depend on the host operating system (type 2) for functionality. In the type 2 model, The VMM depends on the host kernel to provide IO, scheduling, and other low-level functionality, potentially exposing the host kernel and side-channels through shared data structures. Williams et al [60] found that virtualization does lead to fewer host kernel functions being called than direct implementation (although more than their libOS-based approach). However, Li et al [32] demonstrate the effectiveness of a 'popular paths' metric, showing that only 3% of kernel bugs are found in frequently-used code paths (which, in our experience, overlap highly with the code paths used by the VMM).

To illustrate this complexity, the popular combination of Linux Kernel Virtual Machine [29] (KVM) and QEMU clearly illustrates the complexity. QEMU contains > 1.4 million lines of code, and can require up to 270 unique syscalls [57] (more than any other package on Ubuntu Linux 15.04). The KVM code in Linux adds another 120,000 lines. The NEMU [24] project aims to cut down QEMU by removing unused features, but appears to be inactive.

Efforts have been made (such as with Muen [9] and Nova [55]) to significantly reduce the size of the Hypervisor and VMM, but none of these minimized solutions offer the platform independence, operational characteristics, or maturity that we needed at AWS.

Firecracker's approach to these problems is to use KVM (for reasons we discuss in section 3), but replace the VMM with a minimal implementation written in a safe language. Minimizing the feature set of the VMM helps reduce surface area, and controls the size of the TCB. Firecracker contains approximately 50k lines of Rust code (96% fewer lines than QEMU), including multiple levels of automated tests, and auto-generated bindings. Intel Cloud Hypervisor [25] takes a similar approach, (and indeed shares much code with Firecracker), while NEMU [24] aims to address these problems by cutting down QEMU.

Despite these challenges, virtualization provides many compelling benefits. From an isolation perspective, the most compelling benefit is that it moves the security-critical interface from the OS boundary to a boundary supported in hardware and comparatively simpler software. It removes the need to trade off between kernel features and security: the guest kernel can supply its full feature set with no change to the threat model. VMMs are much smaller than general-purpose OS kernels, exposing a small number of well-understood abstractions without compromising on software compatibility or requiring software to be modified.

3 The Firecracker VMM

Firecracker is a Virtual Machine Monitor (VMM), which uses the Linux Kernel's KVM virtualization infrastructure to provide minimal virtual machines (MicroVMs), supporting modern Linux hosts, and Linux and OSv guests. Firecracker provides a REST based configuration API; device emulation for disk, networking and serial console; and configurable rate limiting for network and disk throughput and request rate. One Firecracker process runs per MicroVM, providing a simple model for security isolation.

Our other philosophy in implementing Firecracker was to rely on components built into Linux rather than re-implementing our own, where the Linux components offer the right features, performance, and design. For example we pass block IO through to the Linux kernel, depend on Linux's process scheduler and memory manager for handling contention between VMs in CPU and memory, and we use TUN/TAP virtual network interfaces. We chose this path for two reasons. One was implementation cost: high-quality operating system components, such as schedulers, can take decades to get right, especially when they need to deal with multi-tenant workloads on multi-processor machines. The implementations in Linux, while not beyond criticism [36], are well-proven in high-scale deployments.

The other reason was operational knowledge: within AWS, our operators are highly experienced at operating, automating, optimizing Linux systems (for example, Brendan Gregg's books on Linux performance [18] are popular among Amazon teams). Using KVM in Linux, along with the standard Linux programming model, allows our operators to use most of the tools they already know when operating and troubleshooting Firecracker hosts and guests. For example, running *ps* on a Firecracker host will include all the MicroVMs on the host in the process list, and tools like *top*, *vmstat* and even *kill* work as operators expect. While we do not routinely provide access to in-production Firecracker hosts to operators, the ability to use the standard Linux toolset has proven invaluable during the development and testing of our services. In pursuit of this philosophy, Firecracker does sacrifice portability between host operating systems, and inherits a larger trusted compute base.

In implementing Firecracker, we started with Google's Chrome OS Virtual Machine Monitor *crosvm*, re-using some of its components. Consistent with the Firecracker philosophy, the main focus of our adoption of *crosvm* was removing code: Firecracker has fewer than half as many lines of code as *crosvm*. We removed device drivers including USB and GPU, and support for the 9p filesystem protocol. Firecracker and *crosvm* have now diverged substantially. Since diverging from *crosvm* and deleting the unneeded drivers, Firecracker has added over 20k lines of new code, and changed 30k lines. The rust-vmm project[3] maintains a common set of open-source Rust crates (packages) to be shared by Firecracker and crosvm and used as a base by future VMM implementers.

3.1 Device Model

Reflecting its specialization for container and function workloads, Firecracker provides a limited number of emulated devices: network and block devices, serial ports, and partial i8042 (PS/2 keyboard controller) support. For comparison, QEMU is significantly more flexible and more complex, with support for more than 40 emulated devices, including USB, video and audio devices. The serial and i8042 emulation implementations are straightforward: the i8042 driver is less than 50 lines of Rust, and the serial driver around 250. The network and block implementations are more complex, reflecting both higher performance requirements and more inherent complexity. We use *virtio* [40, 48] for network and block devices, an open API for exposing emulated devices from hypervisors. *virtio* is simple, scalable, and offers sufficiently good overhead and performance for our needs. The entire *virtio* block implementation in Firecracker (including MMIO and data structures) is around 1400 lines of Rust.

We chose to support block devices for storage, rather than filesystem passthrough, as a security consideration. Filesystems are large and complex code bases, and providing only block IO to the guest protects a substantial part of the host kernel surface area.

3.2 API

The Firecracker process provides a REST API over a Unix socket, which is used to configure, manage and start and stop MicroVMs. Providing an API allows us to more carefully control the life cycle of MicroVMs. For example, we can start the Firecracker process and pre-configure the MicroVM and only start the MicroVM when needed, reducing startup latency. We chose REST because clients are available for nearly any language ecosystem, it is a familiar model for our targeted developers, and because OpenAPI allows us to provide a machine- and human-readable specification of the API. By contrast, the typical Unix approach of command-line arguments do not allow messages to be passed to the process after it is created, and no popular machine-readable standard exists for specifying structured command-line arguments. Firecracker users can interact with the API using an HTTP client in their language of choice, or from the command line using tools like *curl*.

REST APIs exist for specifying the guest kernel and boot arguments, network configuration, block device configuration, guest machine configuration and cpuid, logging, metrics, rate limiters, and the metadata service. Common defaults are provided for most configurations, so in the simplest use only the guest kernel and one (root) block device need to be configured before the VM is started.

To shut down the MicroVM, it is sufficient to kill the Firecracker process, or issue a *reboot* inside the guest. As with the rest of Firecracker, the REST API is intentionally kept simple and minimal, especially when compared to similar APIs like Xen's Xenstore.

3.3 Rate Limiters, Performance and Machine Configuration

The machine configuration API allows hosts to configure the amount of memory and number of cores exposed to a MicroVM, and set up the *cpuid* bits that the MicroVM sees. While Firecracker offers no emulation of missing CPU functionality, controlling *cpuid* allows hosts to hide some of their capabilities from MicroVMs, such as to make a heterogeneous compute fleet appear homogeneous.

Firecracker's block device and network devices offer built-in rate limiters, also configured via the API. These rate limiters allow limits to be set on operations per second (IOPS for disk, packets per second for network) and on bandwidth for each device attached to each MicroVM. For the network, separate limits can be set on receive and transmit traffic. Limiters are implemented using a simple in-memory token bucket, optionally allowing short-term bursts above the base rate, and a one-time burst to accelerate booting. Having rate limiters

[3] https://github.com/rust-vmm/community

be configurable via the API allows us to vary limits based on configured resources (like the memory configured for a Lambda function), or dynamically based on demand. Rate limiters play two roles in our systems: ensuring that our storage devices and networks have sufficient bandwidth available for control plane operations, and preventing a small number of busy MicroVMs on a server from affecting the performance of other MicroVMs.

While Firecracker's rate limiters and machine configuration provide the flexibility that we need, they are significantly less flexible and powerful than Linux *cgroups* which offer additional features including CPU credit scheduling, core affinity, scheduler control, traffic prioritization, performance events and accounting. This is consistent with our philosophy. We implemented performance limits in Firecracker where there was a compelling reason: enforcing rate limits in device emulation allows us to strongly control the amount of VMM and host kernel CPU time that a guest can consume, and we do not trust the guest to implement its own limits. Where we did not have a compelling reason to add the functionality to Firecracker, we use the capabilities of the host OS.

3.4 Security

Architectural and micro-architectural side-channel attacks have existed for decades. Recently, the publication of Meltdown [34], Spectre [30], Zombieload [49], and related attacks (e.g. [2, 37, 54, 58]) has generated a flurry of interest in this area, and prompted the development of new mitigation techniques in operating systems, processors, firmware and microcode. Canella et al [10] and ARM [33] provide good summaries of the current state of research. With existing CPU capabilities, no single layer can mitigate all these attacks, so mitigations need to be built into multiple layers of the system. For Firecracker, we provide clear guidance on current side-channel mitigation best-practices for deployments of Firecracker in production[4]. Mitigations include disabling Symmetric MultiThreading (SMT, aka HyperThreading), checking that the host Linux kernel has mitigations enabled (including Kernel Page-Table Isolation, Indirect Branch Prediction Barriers, Indirect Branch Restricted Speculation and cache flush mitigations against L1 Terminal Fault), enabling kernel options like Speculative Store Bypass mitigations, disabling swap and samepage merging, avoiding sharing files (to mitigate timing attacks like Flush+Reload [61] and Prime+Probe [42]), and even hardware recommendations to mitigate RowHammer [28, 43]. While we believe that all of these practices are necessary in a public cloud environment, and enable them in our Lambda and Fargate deployments of Firecracker, we also recognize that tradeoffs exist between performance and security, and that Firecracker consumers in less-demanding environments may choose not to implement some of these mitigations. As with all security mitigations, this is not an end-point, but an ongoing process of understanding and responding to new threats as they surface.

Other side-channel attacks, such as power and temperature, are not addressed by Firecracker, and instead must be handled elsewhere in the system architecture. We have paid careful attention to mitigating these attacks in our own services, but anybody who adopts Firecracker must understand them and have plans to mitigate them.

3.4.1 Jailer

Firecracker's jailer implements an additional level of protection against unwanted VMM behavior (such as a hypothetical bug that allows the guest to inject code into the VMM). The jailer implements a wrapper around Firecracker which places it into a restrictive sandbox before it boots the guest, including running it in a *chroot*, isolating it in *pid* and network namespaces, dropping privileges, and setting a restrictive *seccomp-bpf* profile. The sandbox's *chroot* contains only the Firecracker binary, */dev/net/tun*, cgroups control files, and any resources the particular MicroVM needs access to (such as its storage image). The *seccomp-bpf* profile whitelists 24 syscalls, each with additional argument filtering, and 30 ioctls (of which 22 are required by KVM ioctl-based API).

4 Firecracker In Production

4.1 Inside AWS Lambda

Lambda [51] is a compute service which runs functions in response to events. Lambda offers a number of built-in language runtimes (including Python, Java, NodeJS, and C#) which allows functions to be provided as snippets of code implementing a language-specific runtime interface. A "Hello, World!" Lambda function can be implemented in as few as three lines of Python or Javascript. It also supports an HTTP/REST runtime API, allowing programs which implement this API to be developed in any language, and provided either as binaries or a bundle alongside their language implementation. Lambda functions run within a sandbox, which provides a minimal Linux userland and some common libraries and utilities. When Lambda functions are created, they are configured with a memory limit, and a maximum runtime to handle each individual event[5]. Events include those explicitly created by calling the Lambda *Invoke* API, from HTTP requests via AWS's Application Load Balancer and API Gateway, and from integrations with other AWS services including storage (S3), queue (SQS), streaming data (Kinesis) and database (DynamoDB) services.

Typical use-cases for AWS Lambda include backends for IoT, mobile and web applications; request-response and event-

[4] https://github.com/firecracker-microvm/firecracker/blob/master/docs/prod-host-setup.md

[5] As of early 2019, Lambda limits memory to less than 3GB and runtime to 15 minutes, but we expect these limits to increase considerably over time.

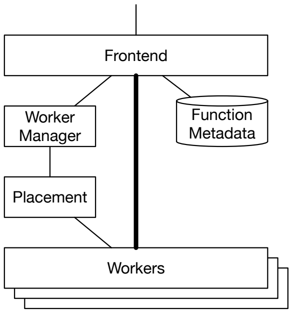


Figure 2: High-level architecture of AWS Lambda event path, showing control path (light lines) and data path (heavy lines)

sourced microservices; real-time streaming data processing; on-demand file processing; and infrastructure automation. AWS markets Lambda as *serverless* compute, emphasizing that Lambda functions minimize operational and capacity planning work, and entirely eliminate per-server operations for most use-cases. Most typical deployments of Lambda functions use them with other services in the AWS suite: S3, SQS, DynamoDB and Elasticache are common companions. Lambda is a large-scale multi-tenant service, serving trillions of events per month for hundreds of thousands of customers.

4.1.1 High-Level Architecture

Figure 2 presents a simplified view of the architecture of Lambda. Invoke traffic arrives at the frontend via the *Invoke* REST API, where requests are authenticated and checked for authorization, and function metadata is loaded. The frontend is a scale-out shared-nothing fleet, with any frontend able to handle traffic for any function. The execution of the customer code happens on the Lambda worker fleet, but to improve cache locality, enable connection re-use and amortize the costs of moving and loading customer code, events for a single function are sticky-routed to as few workers as possible. This sticky routing is the job of the Worker Manager, a custom high-volume (millions of requests per second) low-latency (<10ms 99.9th percentile latency) stateful router. The Worker Manager replicates sticky routing information for a single function (or small group of functions) between a small number of hosts across diverse physical infrastructure, to ensure high availability. Once the Worker Manager has identified which worker to run the code on, it advises the invoke service which sends the payload directly to the worker to reduce round-trips. The Worker Manager and workers also implement a concurrency control protocol which resolves the race conditions created by large numbers of independent invoke services operating against a shared pool of workers.

Each Lambda worker offers a number of *slots*, with each slot providing a pre-loaded execution environment for a function. Slots are only ever used for a single function, and a single concurrent invocation of that function, but are used for many serial invocations of the function. The MicroVM and the process the function is running in are both re-used, as illustrated by Listing 1 which will return a series of increasing numbers when invoked with a stream of serial events.

Listing 1 Lambda function illustrating slot re-use. The returned number will count up over many invokes.

```
var i = 0;
exports.handler = async (event, context) => {
    return i++;
};
```

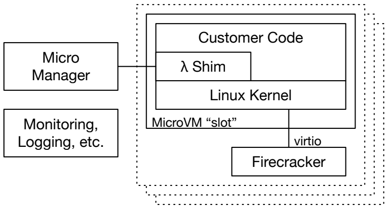


Figure 3: Architecture of the Lambda worker

Where a slot is available for a function, the Worker Manager can simply perform its lightweight concurrency control protocol, and tell the frontend that the slot is available for use. Where no slot is available, either because none exists or because traffic to a function has increased to require additional slots, the Worker Manager calls the Placement service to request that a new slot is created for the function. The Placement service in turn optimizes the placement of slots for a single function across the worker fleet, ensuring that the utilization of resources including CPU, memory, network, and storage is even across the fleet and the potential for correlated resource allocation on each individual worker is minimized. Once this optimization is complete — a task which typically takes less than 20ms — the Placement service contacts a worker to request that it creates a slot for a function. The Placement service uses a time-based lease [17] protocol to lease the resulting slot to the Worker Manager, allowing it to make autonomous decisions for a fixed period of time.

The Placement service remains responsible for slots, including limiting their lifetime (in response to the life cycle of the worker hosts), terminating slots which have become idle or redundant, managing software updates, and other similar activities. Using a lease protocol allows the system to both maintain efficient sticky routing (and hence locality) and have clear ownership of resources. As part of its optimization responsibilities, the placement service also consumes load and health data for each slot in each worker.

4.1.2 Firecracker In The Lambda Worker

Figure 3 shows the architecture of the Lambda worker, where Firecracker provides the critical security boundary required to run a large number of different workloads on a single server.

Each worker runs hundreds or thousands of MicroVMs (each providing a single *slot*), with the number depending on the configured size of each MicroVM, and how much memory, CPU and other resources each VM consumes. Each MicroVM contains a single sandbox for a single customer function, along with a minimized Linux kernel and userland, and a shim control process. The MicroVM is a primary security boundary, with all components assuming that code running inside the MicroVM is untrusted. One Firecracker process is launched per MicroVM, which is responsible for creating and managing the MicroVM, providing device emulation and handling VM exits.

The shim process in each MicroVM communicates through the MicroVM boundary via a TCP/IP socket with the MicroManager, a per-worker process which is responsible for managing the Firecracker processes. MicroManager provides slot management and locking APIs to placement, and an event invoke API to the Frontend. Once the Frontend has been allocated a slot by the WorkerManager, it calls the MicroManager with the details of the slot and request payload, which the MicroManager passes on to the Lambda shim running inside the MicroVM for that slot. On completion, the MicroManager receives the response payload (or error details in case of a failure), and passes these onto the Frontend for response to the customer. Communicating into and out of the MicroVM over TCP/IP costs some efficiency, but is consistent with the design principles behind Firecracker: it keeps the MicroManager process loosely coupled from the MicroVM, and re-uses a capability of the MicroVM (networking) rather than introducing a new device. The MicroManager's protocol with the Lambda shim is important for security, because it is the boundary between the multi-tenant Lambda control plane, and the single-tenant (and single-function) MicroVM. Also, on each worker is a set of processes that provides monitoring, logging, and other services for the worker. Logs and metrics are provided for consumption by both humans and automated alarming and monitoring systems, and metrics are also provided back to Placement to inform its view of the load on the worker.

The MicroManager also keeps a small pool of pre-booted MicroVMs, ready to be used when Placement requests a new slot. While the 125ms boot times offered by Firecracker are fast, they are not fast enough for the scale-up path of Lambda, which is sometimes blocking user requests. Fast booting is a first-order design requirement for Firecracker, both because boot time is a proxy for resources consumed during boot, and because fast boots allow Lambda to keep these spare pools small. The required mean pool size can be calculated with Little's law [35]: the pool size is the product of creation rate and creation latency. Alternatively, at 125ms creation time, one pooled MicroVM is required for every 8 creations per second.

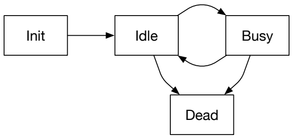


Figure 4: State transitions for a single slots on a Lambda worker

4.2 The Role of Multi-Tenancy

Soft-allocation (the ability for the platform to allocate resources on-demand, rather than at startup) and multi-tenancy (the ability for the platform to run large numbers of unrelated workloads) are critical to the economics of Lambda. Each slot can exist in one of three states: *initializing*, *busy*, and *idle*, and during their lifetimes slots move from *initializing* to *idle*, and then move between *idle* and *busy* as invokes flow to the slot (see Figure 4). Slots use different amounts of resources in each state. When they are *idle* they consume memory, keeping the function state available. When they are *initializing* and *busy*, they use memory but also resources like CPU time, caches, network and memory bandwidth and any other resources in the system. Memory makes up roughly 40% of the capital cost of typical modern server designs, so *idle* slots should cost 40% of the cost of busy slots. Achieving this requires that resources (like CPU) are both soft-allocated and oversubscribed, so can be sold to other slots while one is *idle*.

Oversubscription is fundamentally a statistical bet: the platform must ensure that resources are kept as busy as possible, but some are available to any slot which receives work. We set some compliance goal X (e.g., 99.99%), so that functions are able to get all the resources they need with no contention $X\%$ of the time. Efficiency is then directly proportional to the ratio between the Xth percentile of resource use, and mean resource use. Intuitively, the mean represents revenue, and the Xth percentile represents cost. Multi-tenancy is a powerful tool for reducing this ratio, which naturally drops approximately with $\sqrt{N}$ when running N uncorrelated workloads on a worker. Keeping these workloads uncorrelated requires that they are unrelated: multiple workloads from the same application, and to an extent from the same customer or industry, behave as a single workload for these purposes.

4.3 Experiences Deploying and Operating Firecracker

Starting in 2018, we migrated Lambda customers from our first isolation solution (based on containers per function, and AWS EC2 instances per customer) to Firecracker running on AWS EC2 bare metal instances, with no interruption to availability, latency or other metrics. Each Lambda slot exists for at most 12 hours before it is recycled. Simply changing the recycling logic to switch between Firecracker and legacy implementations allowed workloads to be migrated with no change in behavior.

We took advantage of users of Lambda inside AWS by migrating their workloads first, and carefully monitoring their metrics. Having access to these internal customer's metrics and code reduced the risk of early stages of deployment, because we didn't need to rely on external customers to inform us of subtle issues. This migration was mostly uneventful, but we did find some minor issues. For example, our Firecracker fleet has Symmetric MultiThreading (SMT, aka Hyperthreading) disabled [52] while our legacy fleet had it enabled[6]. Migrating to Firecracker changed the timings of some code, and exposed minor bugs in our own SDK, and in Apache Commons HttpClient [22, 23]. Once we moved all internal AWS workloads, we started to migrate external customers. This migration has been uneventful, despite moving arbitrary code provided by hundreds of thousands of AWS customers.

Throughout the migration, we made sure that the metrics and logs for the Firecracker and legacy stacks could be monitored independently. The team working on the migration carefully compared the metrics between the two stacks, and investigated each case where they were diverged significantly. We keep detailed per-workload metrics, including latency and error-rate, allowing us to use statistical techniques to proactively identify anomalous behavior. Architecturally, we chose to use a common production environment for both stacks as far as possible, isolating the differences between the stacks behind a common API. This allowed the migration team to work independently, shifting the target of the API traffic with no action from teams owning other parts of the architecture.

Our deployment mechanism was also designed to allow fast, safe, and (in some cases) automatic rollback, moving customers back to the legacy stack at the first sign of trouble. Rollback is a key operational safety practice at AWS, allowing us to mitigate customer issues quickly, and then investigate. One case where we used this mechanism was when some customers reported seeing DNS-related performance issues: we rolled them back, and then root-caused the issue to a misconfiguration causing DNS lookups not to be cached inside the MicroVM.

One area of focus for our deployment was building mechanisms to patch software, including Firecracker and the guest and host kernels, quickly and safely and then audit that patches have reached the whole fleet. We use an *immutable infrastructure* approach, where we patch by completely re-imaging the host, implemented by terminating and re-launching our AWS EC2 instances with an updated machine image (AMI). We chose this approach based on a decade of experience operating AWS services, and learning that it is extremely difficult to keep software sets consistent on large-scale mutable fleets (for example, package managers like *rpm* are non-deterministic, producing different results on across a fleet). It also illustrates the value of building higher-level services like Lambda on lower-level services like AWS EC2: we didn't need to build any host management or provisioning infrastructure.

5 Evaluation

In Section 2, we described six criteria for choosing a mechanism for isolation in AWS Lambda: Isolation, Overhead, Performance, Compatibility, Fast Switching and Soft Allocation. In this section we evaluate Firecracker against these criteria as well as other solutions in this space. We use Firecracker v0.20.0 as the base line and use QEMU v4.2.0 (statically compiled with a minimal set of options) for comparison. We also include data for the recently released Intel Cloud Hypervisor [25], a VMM written in Rust sharing significant code with Firecracker, while targeting a different application space.

Our evaluation is performed on an EC2 *m5d.metal* instance, with two Intel Xeon Platinum 8175M processors (for a total of 48 cores with hyper-threading disabled), 384GB of RAM, and four 840GB local NVMe disks. The base OS was Ubuntu 18.04.2 with kernel 4.15.0-1044-aws. The configuration and scripts used for all our experiments, and the resulting data, is publicly available[7].

5.1 Boot Times

Boot times of MicroVMs are important for serverless workloads like Lambda. While Lambda uses a small local pool of MicroVMs to hide boot latency from customers, the costs of switching between workloads (and therefore the cost of creating new MicroVMs) is very important to our economics. In this section we compare the boot times of different VMMs. The boot time is measured as the time between when VMM process is forked and the guest kernel forks its `init` process. For this experiment we use a minimal `init` implementation, which just writes to a pre-configured IO port. We modified all VMMs to call `exit()` when the write to this IO port triggers a VM exit.

All VMMs directly load a Linux 4.14.94 kernel via the command line (i.e. the kernel is directly loaded by the VMM and not some bootloader). The kernel is configured with the settings we recommend for MicroVMs (minimal set of kernel driver and no kernel modules), and we use a file backed minimal root filesystem containing the `init` process. The VMs are configured with a single vCPU and 256MB of memory.

Figure 5 shows the cumulative distribution of (wall-clock) kernel boot times for 500 samples executed serially, so only one boot was taking place on the system at a time. Firecracker results are presented in two ways: end-to-end, including forking the Firecracker process and configuration through the API; and pre-configured where Firecracker has already been set up

[6] We disable SMT on the Firecracker fleet as a sidechannel mitigation. On the legacy fleet the same threats are mitigated by pinning VMs to cores.

[7] https://github.com/firecracker-microvm/nsdi2020-data

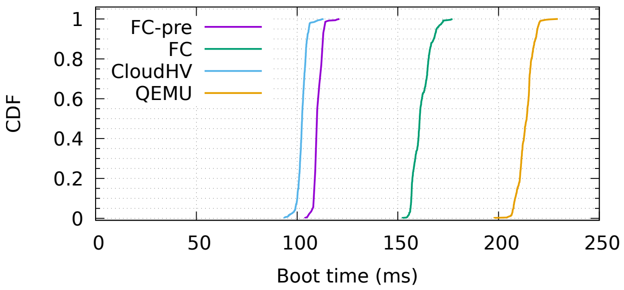


Figure 5: Cumulative distribution of wall-clock times for starting MicroVMs in serial, for pre-configured Firecracker (FC-pre), end-to-end Firecracker, Cloud Hypervisor, and QEMU.

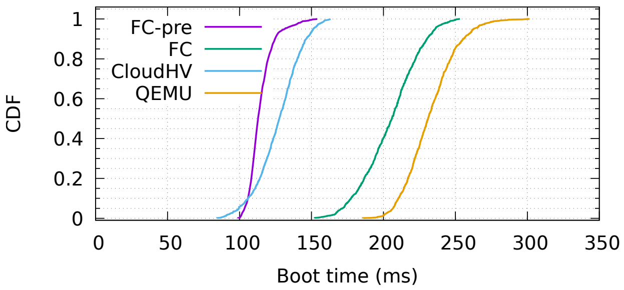


Figure 6: Cumulative distribution of wall-clock times for starting 50 MicroVMs in parallel, for pre-configured Firecracker (FC-pre), end-to-end Firecracker, Cloud Hypervisor,and QEMU.

through the API and time represent the wall clock time from the final API call to start the VM until the `init` process gets executed.

Pre-configured Firecracker and the Cloud Hypervisor perform significantly better than QEMU, both on average and with a tighter distribution, booting twice as fast as QEMU. The end-to-end Firecracker boot times are somewhat in-between, which is expected since we currently perform several API calls to configure the Firecracker VM. Cloud Hypervisor boots marginally faster than pre-configured Firecracker. We suspect subtle differences in the VM emulation to be the reason.

We then booted 1000 MicroVMs with 50 VMs launching concurrently. While any highly-concurrent test will give results that vary between runs, the results in Figure 6 are representative: Firecracker and QEMU perform similarly for end-to-end comparisons, roughly maintaining the 50ms gap already seen with the serial boot times above. Like with serial boot times, pre-configured Firecracker and Cloud Hypervisor show significantly better results, with pre-configured Firecracker out-performing Cloud Hypervisor both in average boot times as well as a tighter distribution with a 99th percentile of 146ms vs 158ms. We also repeated the experiment with starting 100 MicroVMs concurrently and see a similar trend, albeit with a slightly wider distribution with the 99th percentile for pre-configure Firecracker going up to 153ms.

Based on these results, we are investigating a number of optimizations to Firecracker. We could move the Firecracker configuration into a single combined API call would close the gap between the pre-configured and end-to-end boot times. We also investigate to move some of the VM initialization into the VM configuration phase, e.g., allocating the VM memory and loading the kernel image during the configuration API call. This should improve considerably the pre-configured boot times.

The choice of VMM is, of course, only one factor impacting boot times. For example, both Firecracker and Cloud Hypervisor are capable of booting uncompressed Linux kernels while QEMU only boots compressed kernel images. Cloud Hypervisor is also capable of booting compressed kernel. Decompressing the kernel during boot adds around 40ms. The kernel configuration also matters. In the same test setup, the kernel which ships with Ubuntu 18.04 takes an additional 900ms to start! Part of this goes to timeouts when probing for legacy devices not emulated by Firecracker, and part to loading unneeded drivers. In our kernel configuration, we exclude almost all drivers, except the *virtio* and serial drivers that Firecracker supports,build all functionality into the kernel (avoiding modules), and disable any kernel features that typical container and serverless deployments will not need. The compressed kernel built with this configuration is 4.0MB with no modules, compared to the Ubuntu 18.04 kernel at 6.7MB with 44MB of modules. We also recommend a kernel command line for Firecracker which, among other things, disables the standard logging to the serial console (saving up to 70ms of boot time). Note, we use similar optimizations for the other VMMs in iur tests.

For the tests above, all MicroVMs were configured without networking. Adding a statically configured network interface adds around 20ms to the boot times for Firecracker and Cloud Hypervisor and around 35ms for QEMU. Finally, QEMU requires a BIOS to boot. For our test we use *qboot* [8], a minimal BIOS, which reduces boot times by around 20ms compared to the default BIOS. Overall the boot times compare favourable to those reported in the literature, for example [38] reported boot times for an unmodified Linux kernel of 180ms.

5.2 Memory overhead

For Lambda, memory overhead is a key metric, because our goal is to sell all the memory we buy. Lower memory overhead also allows for a higher density of MicroVMs. We measure the memory overhead for the different VMMs as the difference between memory used by the VMM process and the configured MicroVM size. To measure the size of the VMM process we parse the output of the `pmap` command and add up all non-shared memory segments reported. This way our

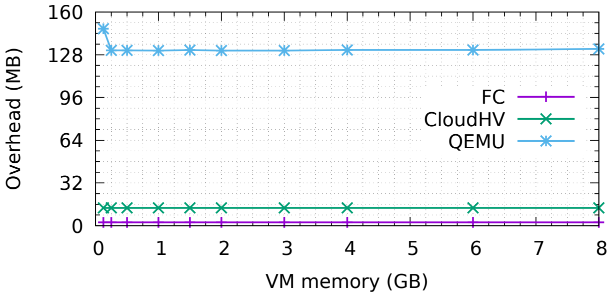


Figure 7: Memory overhead for different VMMs depending on the configured VM size.

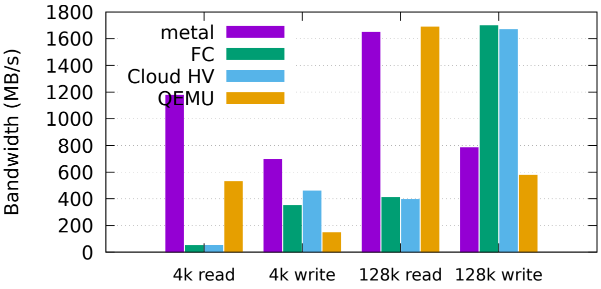


Figure 8: IO throughput on EC2 m5d.metal and running inside various VMs.

calculation excludes the size of the VMM binary itself.

Figure 7 shows that all three VMMs have a constant memory overhead irrespective of the configured VM size (with the exception of QEMU, where for a 128MB sized VM, the overhead is slightly higher). Firecracker has the smallest overhead (around 3MB) of all VMM sizes measured. Cloud Hypervisors overhead is around 13MB per VM. QEMU has the highest overhead of around 131MB per MicroVM. In particular for smaller MicroVMs, QEMU's overhead is significant.

We also measured the memory available inside the MicroVM using the `free` command. The difference in memory available between the different VMMs is consistent with the data presented above.

5.3 IO performance

In this section, we look at the IO performance of the different VMMs both for disks and networking. For all tests we use VMs configured with 2 vCPUs and 512MB of memory. For block IO performance tests we use `fio` [3], and rate limiting was disabled in Firecracker. `fio` is configured to perform random IO directly against the block device, using direct IO through libaio, and all tests were backed by the local NVMe SSDs on the *m5d.metal* server. Figure 8 shows the performance of random read and write IO of small (4kB) and large (128kB) blocks (queue depth 32). The results reflect two limitations in Firecracker's (and Cloud Hypervisor's) current block IO implementation: it does not implement flush-to-disk so high write performance comes at the cost of durability (particularly visible in 128kB write results), and it is currently limited to handling IOs serially (clearly visible in the read results). The hardware is capable of over 340,000 read IOPS (1GB/s at 4kB), but the Firecracker (and Cloud Hypervisor) guest is limited to around 13,000 IOPS (52MB/s at 4kB). We expect to fix both of these limitations in time. QEMU clearly has a more optimized IO path and performs flushing on write. For 128k reads and writes it almost matches the performance of the physical disk, but for 4k operations the higher transaction rate highlights its overheads.

Figure 9 shows 99th percentile latency for block IO with a queue depth of 1. Here Firecracker fares fairly well for small blocks: 4kB reads are only 49µs slower than native reads. Large blocks have significantly more overhead, with Firecracker more than doubling the IO latency. Writes show divergence in the implementation between Firecracker and Cloud Hypervisor with the latter having significantly lower write latency. The write latency of Firecracker and Cloud Hypervisor have also be considered with care since neither supports flush on write.

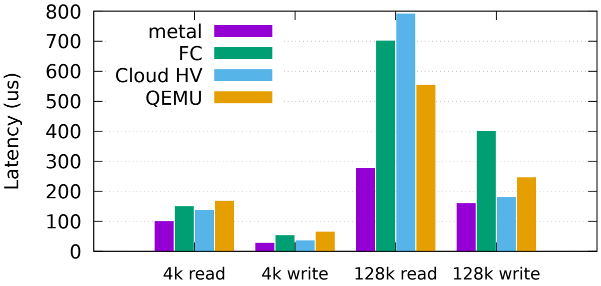


Figure 9: 99th percentile IO latency on EC2 m5d.metal and running inside various VMMs.

Tests with multiple MicroVMs showed no significant contention: running multiple tests each with a dedicated disk, both Firecracker and QEMU saw no degradation relative to a single MicroVM.

Network performance tests were run with *iperf3*, measuring bandwidth to and from the local host over a *tap* interface with 1500 byte MTU. We measured both the performance of a single stream as well as 10 concurrent streams. The results are summarised in Table 1. The host (via the tap interface) can achieve around 44Gb/s for a single stream and 46Gb/s for 10 concurrent streams. Firecracker only achives around 15Gb/s for all scenarios while Cloud Hypervisor achieves slight higher performance, likely due to a slight more optimised virtio implementation. The QEMU throughput is roughly the same as for Could Hypervisor. While Firecracker has a little lower throughput than the other VMMs we have not seen this to be a limitation in our production environment.

As with block IO, performance scaled well, with up to 16 concurrent MicroVMs able to achieve the same bandwidth.

VMM	1 RX	1 TX	10 RX	10 TX
loopback	44.14	44.14	46.92	46.92
FC	15.61	14.15	15.13	14.87
Cloud HV	23.12	20.96	22.53	N/A
Qemu	23.76	20.43	19.30	30.43

Table 1: `iperf3` throughput in Gb/s for receiving (RX) in the VM and transmitting (TX) from the VM for a single and ten concurrent TCP streams.

We expect that there are significant improvements still possible to increase latency and throughput for both disk IO and networking. Some improvements, such as exposing parallel disk IOs to the underlying storage devices, are likely to offer significantly improved performance. Nevertheless, the virtio-based approach we have taken with Firecracker will not yield the near-bare-metal performance offered by PCI pass-through (used in our standard EC2 instances); hardware is not yet up to the task of supporting thousands of ephemeral VMs.

5.4 Does Firecracker Achieve Its Goals?

Using the six criteria from Section 2, we found that while there is scope for improving Firecracker (as there is scope for improving all software), it does meet our goals. We have been running Lambda workloads in production on Firecracker since 2018.

Isolation: The use of virtualization, along with side-channel hardening, implementation in Rust, extensive testing and validation makes us confident to run multiple workloads from multiple tenants on the same hardware with Firecracker.

Overhead and Density: Firecracker is able to run thousands of MicroVMs on the same hardware, with overhead as low a 3% on memory and negligible overhead on CPU.

Performance Block IO and network performance have some scope for improvement, but are adequate for the needs of Lambda and Fargate.

Compatibility: Firecracker MicroVMs run an unmodified (although minimal) Linux kernel and userland. We have not found software that does not run in a MicroVM, other than software with specific hardware requirements.

Fast Switching: Firecracker MicroVMs boot with a production kernel configuration and userland in as little as 150ms, and multiple MicroVMs can be started at the same time without contention.

Soft Allocation: We have tested memory and CPU oversubscription ratios of over 20x, and run in production with ratios as high as 10x, with no issues.

In response to this success, we have deployed Firecracker in production in AWS Lambda, where it is being used successfully to process trillions of events per month for millions of different workloads.

6 Conclusion

In building Firecracker, we set out to create a VMM optimized for serverless and container workloads. We have successfully deployed Firecracker to production in Lambda and Fargate, where it has met our goals on performance, density and security. In addition to the short-term success, Firecracker will be the basis for future investments and improvements in the virtualization space, including exploring new areas for virtualization technology. We are excited to see Firecracker being picked up by the container community, and believe that there is a great opportunity to move from Linux containers to virtualization as the standard for container isolation across the industry.

The future is bright for MicroVMs, both in and out of the cloud. Challenges remain in further optimizing performance and density, building schedulers than can take advantage of the unique capabilities of MicroVM-based isolation, and in exploring alternative operating systems and programming models for serverless computing. We expect that there is much fruitful research to do at the VMM and hypervisor levels. Directions we are interested in include: increasing density (especially memory deduplication) without sacrificing isolation against architectural and microarchitectural side-channel attacks; compute hardware optimized for high-density multi-tenancy; high-density host bypass for networking, storage and accelerator hardware; reducing the size of the virtualization trusted compute base; and dramatically reducing startup and workload switching costs. The hardware, user expectations, and threat landscape around running multitenant container and function workloads are changing fast. Perhaps faster than at any other point in the last decade. We are excited to continue to work with the research and open source communities to meet these challenges.

7 Acknowledgements

The Firecracker team at AWS, and the open source community, made Firecracker and this paper possible. Production experience and feedback from the AWS Lambda and Fargate teams was invaluable. Thanks to Tim Harris, Andy Warfield, Radu Weiss and David R. Richardson for their valuable feedback on the paper.

References

[1] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat

Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 923–935, 2018.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. *IACR Cryptology ePrint Archive*, 2018:1060, 2018.

[3] Jens Axboe. Fio: Flexible i/o tester, 2019. URL: https://github.com/axboe/fio/.

[4] Microsoft Azure. Azure functions, 2019. URL: https://azure.microsoft.com/en-us/services/functions/.

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

[6] Andrew Baumann, Dongyoon Lee, Pedro Fonseca, Lisa Glendenning, Jacob R. Lorch, Barry Bond, Reuben Olinsky, and Galen C. Hunt. Composing os extensions safely and efficiently with bascule. In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 239–252, New York, NY, USA, 2013. ACM. URL: http://doi.acm.org/10.1145/2465351.2465375, doi:10.1145/2465351.2465375.

[7] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, volume 41, page 46, 2005.

[8] Paolo Bonzini. Minimal x86 firmware for booting linux kernels, 2019. URL: https://github.com/bonzini/qboot.

[9] Reto Buerki and Adrian-Ken Rueegsegger. Muen-an x86/64 separation kernel for high assurance. *University of Applied Sciences Rapperswil (HSR), Tech. Rep*, 2013.

[10] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. *arXiv preprint arXiv:1811.05441*, 2018.

[11] Google Cloud. KNative, 2018. URL: https://cloud.google.com/knative/.

[12] Sadjad Fouladi, Francisco Romero, Dan Iter, Qian Li, Shuvo Chatterjee, Christos Kozyrakis, Matei Zaharia, and Keith Winstein. From laptop to lambda: Outsourcing everyday jobs to thousands of transient functional containers. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 475–488, Renton, WA, July 2019. USENIX Association. URL: https://www.usenix.org/conference/atc19/presentation/fouladi.

[13] Sadjad Fouladi, Riad S Wahby, Brennan Shacklett, Karthikeyan Balasubramaniam, William Zeng, Rahul Bhalerao, Anirudh Sivaraman, George Porter, and Keith Winstein. Encoding, fast and slow: Low-latency video processing using thousands of tiny threads. In *NSDI*, pages 363–376, 2017.

[14] The Operstack Foundation. Kata Containers - The speed of containers, the security of VMs, 2017. URL: https://katacontainers.io/.

[15] Google. gvisor: Container runtime sandbox, November 2018. URL: https://github.com/google/gvisor.

[16] Google. Google cloud functions, 2019. URL: https://cloud.google.com/functions/.

[17] C. Gray and D. Cheriton. Leases: An efficient fault-tolerant mechanism for distributed file cache consistency. In *Proceedings of the Twelfth ACM Symposium on Operating Systems Principles*, SOSP '89, pages 202–210, New York, NY, USA, 1989. ACM. URL: http://doi.acm.org/10.1145/74850.74870, doi:10.1145/74850.74870.

[18] Brendan Gregg. *Systems performance: enterprise and the cloud*. Pearson Education, 2014.

[19] Daniel Gruss, Erik Kraft, Trishita Tiwari, Michael Schwarz, Ari Trachtenberg, Jason Hennessey, Alex Ionescu, and Anders Fogh. Page cache attacks. *arXiv preprint arXiv:1901.01161*, 2019. URL: https://arxiv.org/abs/1901.01161.

[20] Vipul Gupta, Swanand Kadhe, Thomas Courtade, Michael W. Mahoney, and Kannan Ramchandran. Oversketched newton: Fast convex optimization for serverless systems, 2019. arXiv:1903.08857.

[21] Joseph M Hellerstein, Jose Faleiro, Joseph E Gonzalez, Johann Schleier-Smith, Vikram Sreekanti, Alexey Tumanov, and Chenggang Wu. Serverless computing: One step forward, two steps back. *arXiv preprint arXiv:1812.03651*, 2018.

[22] Apache HttpClient. Thread interrupt flag leaking when aborting httprequest during connection leasing stage, 2019. URL: https://issues.apache.org/jira/browse/HTTPCLIENT-1958.

[23] Apache HttpCore. Connection leak when aborting httprequest after connection leasing, 2019. URL: https://issues.apache.org/jira/browse/HTTPCORE-567.

[24] Intel. Nemu: Modern hypervisor for the cloud, 2018. URL: https://github.com/intel/nemu.

[25] Intel. Cloud hypervisor, 2019. URL: https://github.com/intel/cloud-hypervisor.

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

[27] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Menezes Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. Technical Report UCB/EECS-2019-3, EECS Department, University of California, Berkeley, Feb 2019. URL: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2019/EECS-2019-3.html.

[28] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of dram disturbance errors. *SIGARCH Comput. Archit. News*, 42(3):361–372, June 2014. URL: https://doi.org/10.1145/2678373.2665726, doi:10.1145/2678373.2665726.

[29] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. kvm: The linux virtual machine monitor. In *In Proc. 2007 Ottawa Linux Symposium (OLS '07)*, 2007.

[30] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[31] James Larisch, James Mickens, and Eddie Kohler. Alto: lightweight vms using virtualization-aware managed runtimes. In *Proceedings of the 15th International Conference on Managed Languages & Runtimes*, page 8. ACM, 2018.

[32] Yiwen Li, Brendan Dolan-Gavitt, Sam Weber, and Justin Cappos. Lock-in-pop: Securing privileged operating system kernels by keeping on the beaten path. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 1–13, Santa Clara, CA, 2017. USENIX Association. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/li-yiwen.

[33] Arm Limited. Cache speculation side-channels, version 2.4. Technical report, October 2018. URL: https://developer.arm.com/support/arm-security-updates/speculative-processor-vulnerability/download-the-whitepaper.

[34] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[35] John DC Little. A proof for the queuing formula: L = λ w. *Operations research*, 9(3):383–387, 1961.

[36] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. The linux scheduler: a decade of wasted cores. In *Proceedings of the Eleventh European Conference on Computer Systems*, page 1. ACM, 2016.

[37] Giorgi Maisuradze and Christian Rossow. ret2spec: Speculative execution using return stack buffers. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2109–2122. ACM, 2018.

[38] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, pages 218–233, New York, NY, USA, 2017. ACM. URL: http://doi.acm.org/10.1145/3132747.3132763, doi:10.1145/3132747.3132763.

[39] Ross McIlroy, Jaroslav Sevcík, Tobias Tebbi, Ben L. Titzer, and Toon Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *CoRR*, abs/1902.05178, 2019. URL: http://arxiv.org/abs/1902.05178, arXiv:1902.05178.

[40] OASIS. Virtual i/o device (virtio) version 1.0, March 2016.

[41] OpenFaaS. OpenFaaS: Serverless Functions Made Simple , 2019. URL: https://www.openfaas.com/.

[42] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of aes. In *Proceedings of the 2006 The Cryptographers' Track at the RSA Conference on Topics in Cryptology*, CT-RSA'06, pages 1–20, Berlin, Heidelberg, 2006. Springer-Verlag. URL: http://dx.doi.org/10.1007/11605805_1, doi:10.1007/11605805_1.

[43] K. Park, S. Baeg, S. Wen, and R. Wong. Active-precharge hammering on a row induced failure in ddr3 sdrams under 3nm technology. In *2014 IEEE International Integrated Reliability Workshop Final Report (IIRW)*, pages 82–85, Oct 2014. doi:10.1109/IIRW.2014.7049516.

[44] Matthew Perron, Raul Castro Fernandez, David DeWitt, and Samuel Madden. Starling: A scalable query engine on cloud function services, 2019. arXiv:1911.11727.

[45] Donald E. Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C. Hunt. Rethinking the library os from the top down. *SIGARCH Comput. Archit. News*, 39(1):291–304, March 2011. URL: http://doi.acm.org/10.1145/1961295.1950399, doi:10.1145/1961295.1950399.

[46] The Chromium Projects. Site isolation design, 2018. URL: https://www.chromium.org/developers/design-documents/site-isolation.

[47] Qifan Pu, Shivaram Venkataraman, and Ion Stoica. Shuffling, fast and slow: Scalable analytics on serverless infrastructure. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 193–206, Boston, MA, 2019. USENIX Association. URL: https://www.usenix.org/conference/nsdi19/presentation/pu.

[48] Rusty Russell. Virtio: Towards a de-facto standard for virtual i/o devices. *SIGOPS Oper. Syst. Rev.*, 42(5):95–103, July 2008. URL: http://doi.acm.org/10.1145/1400097.1400108, doi:10.1145/1400097.1400108.

[49] Michael Schwarz, Moritz Lipp, Daniel Moghimi, Jo Van Bulck, Julian Stecklina, Thomas Prescher, and Daniel Gruss. Zombieload: Cross-privilege-boundary data sampling. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 753–768, New York, NY, USA, 2019. Association for Computing Machinery. URL: https://doi.org/10.1145/3319535.3354252, doi:10.1145/3319535.3354252.

[50] Amazon Web Services. AWS Fargate - Run containers without managing servers or clusters, 2018. URL: https://aws.amazon.com/fargate/.

[51] Amazon Web Services. Aws lambda - serverless compute, 2018. URL: https://aws.amazon.com/lambda/.

[52] Amazon Web Services. Firecracker production host setup recommendations, 2018. URL: https://github.com/firecracker-microvm/firecracker/blob/master/docs/prod-host-setup.md.

[53] Vaishaal Shankar, Karl Krauth, Qifan Pu, Eric Jonas, Shivaram Venkataraman, Ion Stoica, Benjamin Recht, and Jonathan Ragan-Kelley. numpywren: serverless linear algebra, 2018. arXiv:1810.09679.

[54] Julian Stecklina and Thomas Prescher. Lazyfp: Leaking fpu register state using microarchitectural side-channels, 2018. arXiv:1806.07480.

[55] Udo Steinberg and Bernhard Kauer. Nova: A microhypervisor-based secure virtualization architecture. In *Proceedings of the 5th European Conference on Computer Systems*, EuroSys '10, pages 209–222, New York, NY, USA, 2010. ACM. URL: http://doi.acm.org/10.1145/1755913.1755935, doi:10.1145/1755913.1755935.

[56] Chia-Che Tsai, Kumar Saurabh Arora, Nehal Bandi, Bhushan Jain, William Jannen, Jitin John, Harry A. Kalodner, Vrushali Kulkarni, Daniela Oliveira, and Donald E. Porter. Cooperation and security isolation of library oses for multi-process applications. In *Proceedings of the Ninth European Conference on Computer Systems*, EuroSys '14, pages 9:1–9:14, New York, NY, USA, 2014. ACM. URL: http://doi.acm.org/10.1145/2592798.2592812, doi:10.1145/2592798.2592812.

[57] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E. Porter. A study of modern linux api usage and compatibility: What to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 16:1–16:16, New York, NY, USA, 2016. ACM. URL: http://doi.acm.org/10.1145/2901318.2901341, doi:10.1145/2901318.2901341.

[58] Ofir Weisse, Jo Van Bulck, Marina Minkin, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Raoul Strackx, Thomas F Wenisch, and Yuval Yarom. Foreshadow-ng: Breaking the virtual memory abstraction with transient out-of-order execution. Technical report, Technical report, 2018.

[59] Dan Williams and Ricardo Koller. Unikernel monitors: Extending minimalism outside of the box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, Denver,

CO, 2016. USENIX Association. URL: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/williams.

[60] Dan Williams, Ricardo Koller, and Brandon Lum. Say goodbye to virtualization for a safer cloud. In *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 18)*, 2018.

[61] Yuval Yarom and Katrina Falkner. Flush+reload: A high resolution, low noise, l3 cache side-channel attack. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 719–732, San Diego, CA, August 2014. USENIX Association. URL: https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.