Firecracker: Lightweight Virtualization for Serverless Applications

Dineshchandar Ravichandran Will Sherrer Karthik Nedunchezhiyan

1 Abstract

Serverless containers and functions are commonly used in the cloud to deploy and manage applications. Their popularity stems from lower operational costs, better hardware utilization, and faster scaling than traditional deployment methods. Because of its extensive use, serverless implementation has a wealth of possibilities when it comes to selecting and implementing it. Virtualization with solid security and significant overhead is a prevalent issue that consumers must pick between—alternatively, container technologies with lower security and lower overhead.

The Firecracker micro VM was built to overcome this issue, which in most circumstances leads to an unacceptable compromise. Firecracker is an open-source Virtual Machine Monitor (VMM) designed for serverless workloads. However, it may also be used for containers, functions, and other computational workloads that fit within a set of limitations. In the next section, we'll look at the design goals and issue a statement that Firecracker's creators came up with, as well as how they came up with a solution for it. We'll also reproduce it in our own system to see its performance.

2 Background

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 solid security and high overhead and container technologies with weaker security and minimal overhead.

This trade-off is unacceptable to public infrastructure providers, who need both strong security and minimal overhead. To meet this need, we will be exploring the 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, which can provide the same level of security as a traditional VM and at the same time able to launch quicker than conventional VMs.

3 Introduction

Serverless computing is becoming more popular as a strategy for distributing and managing software and services in both public cloud and on-premises environments like Microsoft Azure [6], Google cloud functions [5], Amazon Web Services Lambda [4]. The serverless architecture is appealing for various reasons, including reduced time spent managing servers and capacity, automatic scalability, pay-per-use pricing, and integrations with an event and streaming data sources. Containers, most notably Docker, have grown in popularity for similar reasons, including lower operational costs and easier management.

Multitenancy allows servers to be shared across a large number of workloads, and the ability to provide additional functions and containers in milliseconds allows capacity to be swapped between workloads quickly as demand changes. Despite its financial benefits, multitenancy poses substantial hurdles in isolating workloads from one another.

Workloads must be segregated for security (so that one workload cannot access or infer data from another workload) and operational reasons (so that the noisy neighbor effect of one workload does not slow down other workloads). Cloud instance providers (such as AWS EC2) have comparable difficulties, which they have addressed by utilizing hypervisor-based virtualization (such as QEMU/KVM [7, 29] or Xen) or by avoiding multi-tenancy and providing bare-metal instances.

Many more workloads may be executed on a single system with serverless and container models than with

traditional instance models, which increases the economic benefits of multi-tenancy while also increasing the overhead necessary for isolation.

By relying on isolation features built into the Linux kernel, typical Linux container deployments, such as those using Docker and LXC, are lightweight and rapid to deploy. Control groups (cgroups) offer process grouping, resource throttling, and accounting; namespaces split Linux kernel resources like process IDs (PIDs) into namespaces, and seccomp-bpf limits access to syscalls are among these techniques. These techniques provide a formidable toolkit for isolating containers when used together, but their reliance on a single operating system kernel means that security and code compatibility is critical trade-offs.

Container implementors can increase security by limiting syscalls, but this will break programs that rely on the limited calls. This creates difficult tradeoffs: serverless and container service implementors must pick between hypervisor-based virtualization (and the possibly intolerable overhead that comes with it) and Linux containers (and the related compatibility vs. security tradeoffs). According to the authors, the Firecracker VM addresses these problems.

FirecrackerKVM is present, but QEMU completely replaces it in order to create a new Virtual Machine Monitor (VMM), device model, and API for controlling and configuring MicroVMs. Together with KVM, Firecracker creates a new foundation for implementing container and function isolation. It has less than 5MB memory overhead per container and boots to application code in less than 125ms. It can create up to 150 MicroVMs per second using the specified basic Linux guest kernel configuration.

Its able to offer these performance because of the following:

- It does not offer a BIOS, cannot boot arbitrary kernels, does not emulate legacy devices nor PCI, and does not support VM migration.
- It doesn't offer VM orchestration and metadata management.
- Lower-level features, such as additional devices (USB, PCI, sound, video, etc), BIOS, and CPU instruction emulation are also not available, because they are not needed by typical serverless container and function workloads.

4 Related Work:

Over time, a lot of research has been done on serverless containers, VMs, and their applications. With its increasing performance and efficiency, there have been other systems developed that are similar to Firecracker. Other notable applications include QEMU, Cloud-HyperVisor, Unikernel, and LightVM. Firecracker does not offer a BIOS, cannot boot arbitrary kernels, does not emulate legacy devices nor PCI, and does not support VM migration. This is very different from QEMU, which does offer those packages. QEMU [7] is more robust and can work with Kernel-based Virtual Machines (KVM) to run virtual machines at near-application speeds.

Firecracker replaces QEMU, rather than Docker or Kubernetes, in the container stack, as Firecracker was designed to be simplistic and minimalistic. QEMU can also emulate for user-level processes, allowing applications compiled for one architecture to run on another. One very similar application to Firecracker is Cloud-Hypervisor, and the authors compared Firecracker's performance.

Cloud-Hypervisor [8] is a Virtual Machine Monitor (VMM) implemented in Rust that focuses on running modern cloud workloads with minimal hardware emulation. Similar to the Firecracker, it was designed to be secure, lightweight, and fast. Unikernels [3] are another cloud-focused architecture in serverless computing.

Unikernel takes a different approach and provides specialized, single-address-space machine images constructed using library operating systems. They were designed to be safe and lightweight while also maintaining performance. They work by compiling high-level languages, such as JavaScript and Python, directly into specialized machine images that run directly on a hypervisor. This lightweight functionality allows customers to run their services cheaper and more securely, with finer control than a complete software stack.

5 Methodology:

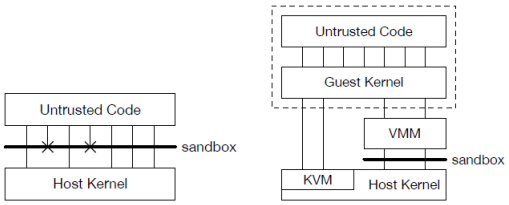
5.1 Choosing an Isolation Solution

In general, there are three types of options for isolating workloads on Linux:

Containers: Containers are systems in which all workloads share a kernel and are isolated via a mixture of kernel methods.

Virtualization: In Virtualization Workloads run in their own virtual machines (VMs) under the control of a hypervisor.

The language virtual machine : Isolation of workloads by language virtual machine is responsible for isolating workloads from each other and from the operating system.



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

Figure 1: Linux container security (a) is based on the kernel's sandboxing capabilities, whereas KVMstyle virtualization (b) is based on the VMM's security, maybe with reinforced sandboxing.

As shown in Figure 1, which contrasts the security techniques used by Linux containers versus virtualization. Untrusted code in Linux containers calls the host kernel directly, possibly with the kernel surface area limited (such as with seccomp-bpf). It also interacts directly with the host kernel's other services, such as filesystems and the page cache.

Un-trusted code is typically given full access to a guest kernel in virtualization, allowing it to exploit all kernel features while explicitly designating the guest kernel as untrusted. The guest kernel's access to the privileged domain and host kernel is limited by hardware virtualization and the VMM.

5.1.1 Linux Containers

Linux containers integrate several Linux kernel technologies to provide operational and security isolation. These features include cgroups, which provide CPU, memory, and other resource limits; namespaces, which provide namespacing for kernel resources such as user IDs (uids), process IDs (pids), and network interfaces; seccomp-bpf, which allows a process to limit which syscalls it can use and which arguments it can pass to these syscalls; and chroot, which provides an isolated filesystem. These tools are used in various combinations by different Linux container implementations, although seccomp-bpf is the most essential security isolation border. Container security relies on syscall limits, which represents a choice between security and compatibility.

It's possible to reduce the kernel surface significantly, especially since it's plausible to suppose that the Linux kernel contains more flaws in syscalls that aren't utilized as much [12]. One solution to this problem is to provide certain operating system capability in userspace, which requires a far smaller amount of kernel code to give the programmer the impression of a fully functional environment.

Container isolation is concerned not only with preventing privilege escalation in contexts running untrusted programs but also with preventing information exposure side channels and communication between functions across covert channels.

5.1.2 Language-Specific Isolation

Language-specific isolation techniques were not suitable for Firecracker,as it was designed to support AWS Lambda / Fargate and their need to support arbitrary binaries.

5.1.3 Virtualization

Virtualization nowadays makes use of hardware characteristics like Intel VT-x to offer each sandbox with its own virtual hardware, page tables, and operating system kernel. Density and overhead are two fundamental virtualization difficulties that are intertwined.

Before it can conduct productive work, the VMM and kernel associated with each guest occupy some CPU and memory, limiting density.

Another issue is starting time, which is typically in the range of seconds for virtual machines.

The deployment of virtualization presents the third challenge: hypervisors and virtual machine monitors (VMMs), and thus the requisite trusted computing base (TCB), can be vast and sophisticated, with a broad attack surface.

Despite these limitations, virtualization has a lot of advantages. The most convincing benefit from the standpoint of isolation is that it moves the security-critical interface from the OS boundary to a hardware-supported and comparatively more straightforward software border. It eliminates the need to choose between kernel functionality and security: the guest kernel can provide its entire feature set while the threat model remains unchanged.

VMMs are substantially smaller than general-purpose OS kernels, exposing only a few well-understood abstractions while preserving software compatibility and avoiding the need to modify applications.

5.2 Device Model of Firecracker VMM

Firecracker emulates a small number of devices, including network and block devices, serial ports, and partial support for the i8042 (PS/2 keyboard controller). QEMU, on the other hand, is much more flexible and complex, supporting over 40 emulated devices, including USB, video, and audio devices.

As a security consideration, Firecracker supports block devices for storage rather than filesystem passthrough. Because filesystems are huge and complicated codebases, limiting the guest to block IO protects a significant portion of the host kernel surface area.

5.3 API connectivity for Firecracker

The Firecracker process provides a REST API that may be used to configure, manage, and start and stop MicroVMs through a Unix socket. We can better regulate the life cycle of MicroVMs by providing an API. Users can, for example, launch the Firecracker process and pre-configure the MicroVM, then start it only when it is needed, decreasing startup latency.

Firecracker uses REST because clients are available for nearly any language ecosystem. 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.

5.4 Rate Limiters, Performance and Machine Configuration

The machine configuration API allows hosts to define the amount of RAM and cores available to a MicroVM, as well as the cpuid bits that the MicroVM sees. Controlling cpuid will enable hosts to disguise parts of their capabilities from MicroVMs, such as making 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 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 be configurable via the API will enable us to vary limits based on configured resources (like the memory configured for a Lambda function) or dynamically based on demand.

In Firecracker, rate limiters serve two functions:

- Assuring that our storage devices and networks have enough bandwidth to support control plane operations.
- Preventing the performance of additional MicroVMs on a server from being impacted by a small number of busy MicroVMs

While Firecracker's rate limiters and machine configuration provide us with the flexibility we require, they are far less flexible and powerful than Linux cgroups, which include features such as CPU credit scheduling, core affinity, scheduler control, traffic prioritization, performance events, and accounting. This is in line with our way of thinking.

Authors established performance restrictions in Firecracker because of this compelling reason; establishing rate limits in device emulation allows us to tightly manage how much VMM and host kernel CPU time a guest can spend, and we don't trust the guest to set its own limits.

5.5 Security

Architectural and micro-architectural side-channel attacks mitigations include disabling Symmetric Multi-Threading (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 same page merging, avoiding sharing files (to mitigate timing attacks like Flush+Reload and Prime+Probe), and even hardware recommendations to mitigate RowHammer [13][11]

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 [1] 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 sandboxes 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).

6 Firecracker In Production

6.1 Firecracker In The Lambda Worker

Since the Firecracker was developed predominantly for AWS, it was evaluated in the AWS Lambda service.

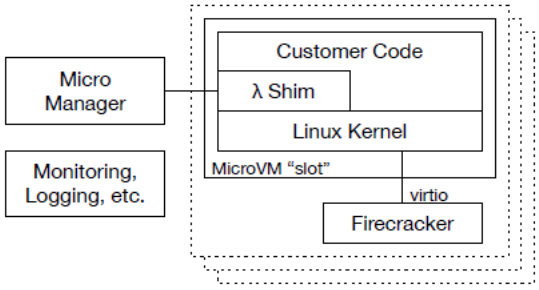


Figure 2 : Architecture of the Lambda worker

Above figure Fig- 2 illustrates the Lambda worker's architecture, in which Firecracker provides the critical security boundary needed to operate a large number of

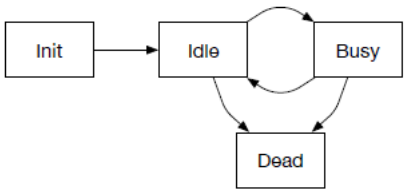
distinct workloads on a single server. Each worker runs hundreds or thousands of MicroVMs (each giving a single slot), the number varying based on the MicroVM's chosen size and the amount of memory, CPU, and other resources it requires. Each MicroVM includes a single sandbox for a single customer function, as well as a Linux kernel and userland that has been minimized and a shim control process.

All components assume that the code running inside the MicroVM is untrusted, making the MicroVM a primary security boundary. Each MicroVM has its own Firecracker process, which is in charge of establishing and administering the MicroVM, as well as providing device emulation and handling VM exits.

6.2 The Role of Multi-Tenancy

The economics of Lambda are dependent on soft-allocation (the ability for the platform to assign resources on-demand rather than at startup) and multi-tenancy (the ability for the platform to execute large numbers of unrelated workloads).

Each slot can be in one of three states: initializing, busy, or idle; during their lifespan, slots go from initializing to idle, then between idle and busy as flow is invoked, as illustrated in the below image.



In each state, different amounts of resources are used by slots. They use memory when they are idle, keeping the function state available. They use memory, as well as other resources such as CPU time, caches, network, memory bandwidth, and any other resources in the system when they are initializing and busy.

Oversubscription is essentially a statistical bet: the platform has to keep resources as busy as possible, but some are accessible to any slot that gets work. We define a compliance objective X (for example, 99.99 percent) to ensure that functions have all the resources they require with no conflict X percent of the time. The ratio between the Xth percentile of resource consumption and the mean resource use is then directly proportional to efficiency. The mean indicates revenue, whereas the Xth percentile represents cost, intuitively. When performing N uncorrelated workloads on a worker, this ratio naturally declines with $\sqrt{N}$. Multi-tenancy is a strong method for decreasing this ratio.

7 Product Targets:

The Firecracker VM was designed to provide strong security against a broad range of attacks (including micro-architectural side-channel attacks), the ability to run at high densities with little overhead or waste, and compatibility with a broad range of unmodified software.

The properties focused to achieve these are:

- **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.
- **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.

8 Implementation

8.1 configure-kernel.sh

```
arch=`uname -m`
kernel_path=$(pwd)"/firecracker/hello-vmlinux.bin"

if [ ${arch} = "x86_64" ]; then
    curl --unix-socket /tmp/firecracker.socket -i \
      -X PUT 'http://localhost/boot-source'   \
      -H 'Accept: application/json'           \
      -H 'Content-Type: application/json'     \
      -d "{
            \"kernel_image_path\": \"${kernel_path}\",
            \"boot_args\": \"console=ttyS0 reboot=k panic=1 pci=off\"
      }"
elif [ ${arch} = "aarch64" ]; then
    curl --unix-socket /tmp/firecracker.socket -i \
      -X PUT 'http://localhost/boot-source'   \
      -H 'Accept: application/json'           \
      -H 'Content-Type: application/json'     \
      -d "{
            \"kernel_image_path\": \"${kernel_path}\",
            \"boot_args\": \"keep_bootcon console=ttyS0 reboot=k panic=1 pci=off\"
      }"
else
    echo "Cannot run firecracker on $arch architecture!"
    exit 1
fi
```

The **configure-kernel.sh** script uses the UNIX domain socket exposed by firecracker to configure our custom Linux kernel. The code will detect the CPU architecture of the machine and loads the different Linux kernel for both $x86-64$ (covers Intel/AMD chipset) and aarch64

(covers ARM chipset). Other CPU architectures are officially not supported by the firecracker.

8.2 configure-rootfs.sh

```
rootfs_path=$(pwd)"/firecracker/hello-rootfs.ext4"
curl --unix-socket /tmp/firecracker.socket -i \
  -X PUT 'http://localhost/drives/rootfs' \
  -H 'Accept: application/json'           \
  -H 'Content-Type: application/json'     \
  -d "{
        \"drive_id\": \"rootfs\",
        \"path_on_host\": \"${rootfs_path}\",
        \"is_root_device\": true,
        \"is_read_only\": false
   }"
~
```

Rootfs is the root file system that will contain the modules, libraries, binaries, and configurations based on which the Linux OS and the user application run. We have configured it with ".ext4" [2] file type, as this is more configurable and the latest file system, which gives us the most flexibility for most of the application scenario.

The ext4 filesystem can support volumes with sizes up to 1 exbibyte (EiB) and single files with sizes up to 16 tebibytes (TiB) with the standard 4 KiB block size.[13] The maximum file, directory, and filesystem size limits grow at least proportionately with the filesystem block size up to the maximum 64 KiB block size available on ARM and PowerPC/Power ISA CPUs, and ext4 uses a performance technique called allocate-on-flush, also known as delayed allocation. That is, ext4 delays block allocation until data is flushed to disk; in contrast, some file systems allocate blocks immediately, even when the data goes into a write cache. Delayed allocation improves performance and reduces fragmentation by effectively allocating larger amounts of data at a time.

8.3 launch-firecracker.sh

```
curl --unix-socket /tmp/firecracker.socket -i \
  -X PUT 'http://localhost/actions'       \
  -H  'Accept: application/json'          \
  -H  'Content-Type: application/json'    \
  -d '{
      "action_type": "InstanceStart"
   }'
~
~
```

This bash script contains the launching instruction to initiate firecracker via launch REST API.

8.4 initial-setup.sh

```
#!/bin/bash

COUNT="${1:-4000}" # Default to 4000

RES=scripts

rm -rf output
mkdir output
chown -R ec2-user:ec2-user output

pushd $RES > /dev/null

./one-time-setup.sh
./setup-network-taps.sh 0 $COUNT 100

popd > /dev/null
~
```

This script will allow the user to configure the no. of Firecracker VMs they would like to launch in parallel, along with the network configuration and one-time setup. These scripts will be discussed in detail in the following sections.

For the above-illustrated example, we have configured the script to launch a max of 4000 VMs. However, in our network setup, we have configured only for 100 instances as we will spin up only 100 VMs for our project.

8.5 one-time-setup.sh

```
# Load kernel module
sudo modprobe kvm_intel

# Configure packet forwarding
sudo sysctl -w net.ipv4.conf.all.forwarding=1

# Avoid "neighbour: arp_cache: neighbor table overflow!"
sudo sysctl -w net.ipv4.neigh.default.gc_thresh1=1024
sudo sysctl -w net.ipv4.neigh.default.gc_thresh2=2048
sudo sysctl -w net.ipv4.neigh.default.gc_thresh3=4096


# Download relevant resources.
./ensure_resources.sh
~
```

This script will update the default threshold for network configuration so multiple VM's can be initiated. And it will also allow the packages to flow from all the VM's to the host machine IP without dropping the package.

This script also configures the threshold for arp-cache with three threshold limits, from 1024-4096 Bytes. An ARP cache is a collection of Address Resolution Protocol entries that are created when an IP address is resolved to a MAC address.

8.6 setup-tap-with-id.sh

```
#!/bin/bash -e

TAP_ID="${1:-0}" # Default to 0
TAP_DEV="fc-${TAP_ID}-tap0"

# Setup TAP device that uses proxy ARP
MASK_LONG="255.255.255.252"
MASK_SHORT="/30"
FC_IP="$(printf '169.254.%s.%s' $(((4 * TAP_ID + 1) / 256)) $(((4 * TAP_ID + 1) % 256)))"
TAP_IP="$(printf '169.254.%s.%s' $(((4 * TAP_ID + 2) / 256)) $(((4 * TAP_ID + 2) % 256)))"
FC_MAC="$(printf '02:FC:00:00:%02X:%02X' $((TAP_ID / 256)) $((TAP_ID % 256)))"
ip link del "$TAP_DEV" 2> /dev/null || true
ip tuntap add dev "$TAP_DEV" mode tap
sysctl -w net.ipv4.conf.${TAP_DEV}.proxy_arp=1 > /dev/null
sysctl -w net.ipv6.conf.${TAP_DEV}.disable_ipv6=1 > /dev/null
ip addr add "${TAP_IP}${MASK_SHORT}" dev "$TAP_DEV"
ip link set dev "$TAP_DEV" up

iperf3 -B $TAP_IP -s > /dev/null 2>&1 &
```

In this script, we configure the virtual ports using tap interfaces for all the VMs created. Each tap interface will have four IP address, for the following functions:

- Network:
  It is the first address in the network and is used to identify network segments. All the IP addresses using the same network address part are in the same network segment. Because the network address is the first address in the network, it can not be a random IP address, but it must match with the network mask in a binary view, for the last bits in the network address must be zeros, as long as the mask has zeros.

- Host (Firecracker-VM):
  In our network topology, We have used only one host address (for firecracker VM). All the traffic will be routed to this IP address.

- TAP interface:
  Similar to the IP address assigned to the physical network interface, We have assigned an IP address to this virtual tap device.

- Broadcast:
  The broadcast address is the last address in the network, and it is used for addressing all the nodes in the network at the same time. It means that IP packet, where the destination address is a broadcast address, is sent to all nodes of the IP network. It is essential for remote announcements in the network segment. In some cases, it is used for attacking purposes by hackers or can cause problems in bigger network segments.

This script will also assign a MAC address for each VM. Post which, this script will activate the TAP interface and disable the IPv6, as currently, for our project scope, we don't need IPv6, and in doing so, we will speed up our boot timing.
Post all the above configuration is completed, we will launch "iperf," which will create a network package and send it to the IP address created for the VM. This is performed for network load testing in order to simulate the actual production environment. Iperf is an open-source networking tool used to measure the throughput or performance of a network. It can be used to test TCP and UDP.

8.7 parallel-start-many.sh

```
#!/bin/bash

#Usage
## ./parallel-start-many.sh 0 100 5 # Will start VM#0 to VM#99 5 at a time.

start="${1:-0}"
upperlim="${2:-1}"
parallel="${3:-1}"

echo Start @ `date`.
START_TS=`date +%s%N | cut -b1-13`
for ((i=0; i<parallel; i++)); do
  s=$((i * upperlim / parallel))
  e=$(((i+1) * upperlim / parallel))
  ./start-many.sh $s $e &
  pids[${i}]=$!
done

# wait for all pids
for pid in ${pids[*]}; do
    wait $pid
done

END_TS=`date +%s%N | cut -b1-13`
END_DATE=`date`

total=$((upperlim-start))
delta_ms=$((END_TS-START_TS))
delta=$((delta_ms/1000))
rate=`bc -l <<< "$total/$delta"`

cat << EOL
Done @ $END_DATE.
Started $total microVMs in $delta_ms milliseconds.
MicroVM mutation rate was $rate microVMs per second.
EOL

./extract-times.sh &
```

Parallel start scrip allows us to initialize multiple Firecracker VMs simultaneously for much better parallel launching and work with the shared resources. This script allows us to test the following:

- To test the amount of time taken to launch multiple VMs using the shared resources. For our project we launched 100 VM's in a span of 120sec.

- It also allows us to test the soft allocation properties of the Firecracker herein, we are launching 100 VM's consuming a total of 1.6GB of RAM, but our host machine has only 1 GB RAM.

This parallel start many will initiate the start many scripts, which based on the no.of Firecracker configured to launch will call on the start-firecracker.sh scrip, as illustrated below:

```
#!/bin/bash

#Usage
## sudo ./start.sh 0 100 # Will start VM#0 to VM#99.

start="${1:-0}"
upperlim="${2:-1}"

for ((i=start; i<upperlim; i++)); do
  ./start-firecracker.sh "$i"
done
```

The Start Firecracker script is used for launching each Firecracker, as illustrated below:

```
#!/bin/bash -x
SB_ID="${1:-0}" # Default to sb_id=0

FC_BINARY="$PWD/resources/firecracker"
RO_DRIVE="$PWD/resources/rootfs.ext4"
KERNEL="$PWD/resources/vmlinux"
TAP_DEV="fc-${SB_ID}-tap0"

KERNEL_BOOT_ARGS="init=/sbin/boottime_init panic=1 pci=off reboot=k tsc=reliable quiet 8250.nr_uarts=0 ipv6.disable=1"
#KERNEL_BOOT_ARGS="console=ttyS0 reboot=k panic=1 pci=off nomodules ipv6.disable=1"

API_SOCKET="/tmp/firecracker-sb${SB_ID}.sock"
CURL=(curl --silent --show-error --header "Content-Type: application/json" --unix-socket "${API_SOCKET}" --write-out "HTTP %{http_code}")

curl_put() {
    local URL_PATH="$1"
    local OUTPUT RC
    OUTPUT="$("${CURL[@]}" -X PUT --data @- "http://localhost/${URL_PATH#/}" 2>&1)"
    RC="$?"
    if [ "$RC" -ne 0 ]; then
        echo "Error: curl PUT ${URL_PATH} failed with exit code $RC, output:"
        echo "$OUTPUT"
        return 1
    fi
}

logfile="$PWD/output/fc-sb${SB_ID}-log"
#metricsfile="$PWD/output/fc-sb${SB_ID}-metrics"
metricsfile="/dev/null"

touch $logfile

# Setup TAP device that uses proxy ARP
MASK_LONG="255.255.255.252"
#MASK_SHORT="/30"
FC_IP="$(printf '169.254.%s.%s' $(((4 * SB_ID + 1) / 256)) $(((4 * SB_ID + 1) % 256)))"
TAP_IP="$(printf '169.254.%s.%s' $(((4 * SB_ID + 2) / 256)) $(((4 * SB_ID + 2) % 256)))"
FC_MAC="$(printf '02:FC:00:00:%02X:%02X' $((SB_ID / 256)) $((SB_ID % 256)))"
#ip link del "$TAP_DEV" 2> /dev/null || true
#ip tuntap add dev "$TAP_DEV" mode tap
#sysctl -w net.ipv4.conf.${TAP_DEV}.proxy_arp=1 > /dev/null
#sysctl -w net.ipv6.conf.${TAP_DEV}.disable_ipv6=1 > /dev/null
#ip addr add "${TAP_IP}${MASK_SHORT}" dev "$TAP_DEV"
#ip link set dev "$TAP_DEV" up

KERNEL_BOOT_ARGS="${KERNEL_BOOT_ARGS} ip=${FC_IP}::${TAP_IP}:${MASK_LONG}::eth0:off"

# Start Firecracker API server
rm -f "$API_SOCKET"
"${FC_BINARY}" --api-sock "$API_SOCKET" --id "${SB_ID}" --boot-timer >> $logfile &

sleep 0.015s

# Wait for API server to start
while [ ! -e "$API_SOCKET" ]; do
    echo "FC $SB_ID still not ready..."
    sleep 0.01s
done

curl_put '/logger' <<EOF
{
  "level": "Info",
  "log_path": "$logfile",
  "show_level": false,
  "show_log_origin": false
}
EOF

curl_put '/metrics' <<EOF
{
  "metrics_path": "$metricsfile"
}
EOF

#curl_put '/machine-config' <<EOF
#{
#  "vcpu_count": 1,
#  "mem_size_mib": 128
#}
#EOF

curl_put '/boot-source' <<EOF
{
  "kernel_image_path": "$KERNEL",
  "boot_args": "$KERNEL_BOOT_ARGS"
}
EOF

curl_put '/drives/1' <<EOF
{
  "drive_id": "1",
  "path_on_host": "$RO_DRIVE",
  "is_root_device": true,
  "is_read_only": true
}
EOF
#
#curl_put '/drives/2' <<EOF
#{
#  "drive_id": "2",
#  "path_on_host": "$RW_DRIVE",
#  "is_root_device": false,
#  "is_read_only": false
#}
#EOF

curl_put '/network-interfaces/1' <<EOF
{
  "iface_id": "1",
  "guest_mac": "$FC_MAC",
  "host_dev_name": "$TAP_DEV"
}
EOF

curl_put '/actions' <<EOF
{
  "action_type": "InstanceStart"
}
EOF
```

9 Results Evaluation

9.0.1 Soft Subscription

In line with our project plan, we were able to do background research on Firecracker and successfully launch it. And we were also able to achieve the secondary objective for the midterm presentation of launching 100 VMs parallel with soft allocation, i.e., to allocate 1.6GB of ram were in total ram in the system where VMs were launched had 1GB of ram.

9.0.2 Density Test

We are able to do so because of the property of Firecracker's VM, which allows us to soft allocate the resources. Also, we were able to launch 100 VMs in 120 seconds, which is quicker than traditional VM.

We also did not find Firecracker to consume more resources than it was expected to when testing and did not need extra overhead to maintain these instances.

9.0.3 Security Test

We tested the disk and memory space as per the configuration, and later we used the VMs to consume more resources than permitted in the allocation configuration. And we were able to observe in these instances; that Firecracker killed these VMs.

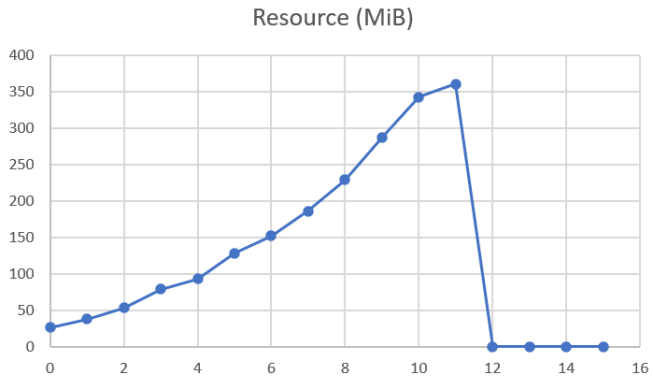


Figure-4: RAM memory consumed in MiB/Sec

As illustrated above in Fig-4, we are able to observe as RAM use-age reached 361 MiB, wherein the configured max cap was 360MiB for that VM. The Firecracker detected this over-usage and killed the VM.

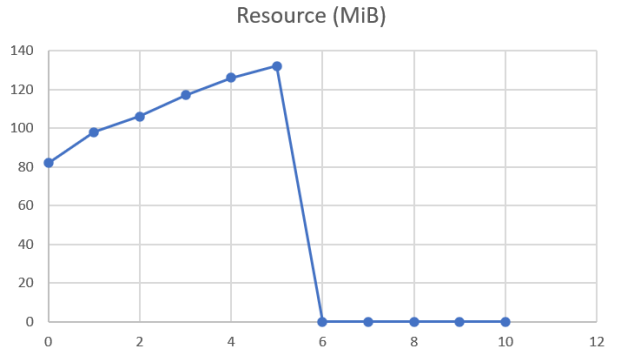


Figure-5: Storage memory consumed in MiB/Sec

As illustrated above in Fig-5, we are able to observe as storage use-age reached 132 MiB, wherein the configured max cap was 135MiB for that VM. The Firecracker detected this over-usage and Killed the VM; in the graph, we are seeing the VM being terminated as it reached 132MiB; this is because the VM reached 135MiB before the next cycle of data tracking.

But in-case of CPU consumption, when more resources were requested on run-time, the firecracker throttled the CPU instead of Killing the VM. As illustrated in below image Figure-6:

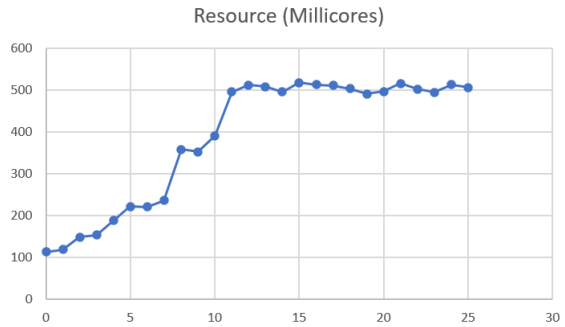


Figure-6: CPU cycles consumed per millicores/second

In this case, we had configured the max cap for core consumption as 500millicores/sec, but as observed, we Firecracker throttled the CPU above 500 millcores/sec. However, it looks like the resource consumption starts plateauing out around 500-520. Still, the firecracker should have killed the VM at 500 millicores/sec consumption.

10 Discrepancies in the paper

- This paper was found to be outdated. Since its inception, there have been multiple updates to Firecracker and the Linux kernel. One such update included the addition of *io$_u$ring* from Linux.

- In Linux's old ways to do file-based I/O, it would use read() and write() syscalls, but the implementation and performance are poor. The most significant failure is its inability to allow for asynchronous I/O regularly. It also suffered from a poor API.

- In 2020 Linux released *io$_u$ring*. It was designed to be an efficient way for asynchronous I/O operations and addressing performance issues with the old interface. Firecracker was redesigned to take advantage of this new implementation since its release.

11 Future Work

- **Memory Ballooning:**
  Essentially, when Firecracker allocates memory to the different VMs, it keeps track of the memory that has been allocated; the host kernel is also aware of the memory that has been allocated. However, when the VM has used the memory and is de-allocating. Firecracker keeps track, but the host kernel is unaware of the de-allocation and still believes the memory is in use. And our suggestion is to handle this via implementing **memory ballooning. [9].**

- **Cgroups:**
  To address the CPU throttling, we propose using Cgroups hooks to terminate the firecracker VM; if the VM utilizes more vCPU than intended. Control groups, usually referred to as cgroups [10], are a Linux kernel feature that allows processes to be organized into hierarchical groups whose usage of various types of resources can then be limited and monitored. The kernel's cgroup interface is provided through a pseudo-filesystem called cgroupfs.

- **Locking the firecracker VM in the PID namespace:**
  To overcome forkbombing, we propose to lock the firecracker VM in the PID namespace by default if the seccomp filter level is set other than strict. PID namespaces isolate the process ID number space, meaning processes in different PID namespaces can have the same PID. PID namespaces allow containers to provide functionality such as suspending/resuming the set of processes in the container and migrating the container to a new host while the functions inside the container maintain the same PIDs

12 Acknowledgements

The Firecracker team, Oracle VM VirtualBox,and the open source community,made it possible for us to recreate and simulate the VM and test its claims to a certain level. We would also like to thank Dr. Long Cheng for his constant guidance and support.

13 Project Repository

The codes we have used for this project are available on: `https://github.com/KarthikNedunchezhiyan/networking-final-demo`

14 Member Contributions

- **All members:** Researching background and implementation of Firecracker.
- **Karthik Nedunchezhiyan:** Writing bash shells scripts for booting and parallel running of Firecracker VM on development VM environment.
- **Dineshchandar Ravichandran:** Writing reports and performing security testing to observe memory consumption.
- **Will Sherrer :** Creating/managing presentations and creating python scripts for analysis VM resources.

References

[1] Linux jailer lambda, 2015.

[2] .ext4, 2016.

[3] Unikernel, 2017.

[4] Aws lambda, 2018.

[5] Google cloud functions, 2019.

[6] Microsoft azure, 2019.

[7] Qemu, 2020.

[8] Cloud hyper-visor, 2021.

[9] Ballooning.

[10] Control groups (cgroups), 2022.

[11] K. PARK, S. BAEG, S. W., AND WONG., R. Active-precharge hammering on a row induced failure in ddr3 sdrams under 3× nm technology. *IEEE Xplore* (2022).

[12] YIWEN LI, BRENDAN DOLAN-GAVITT, S. W. A. J. C. Lock-in-Pop: Securing privileged operating system kernels by keeping on the beaten path. *USENIX* (1970).

[13] YOONGU KIM, ROSS DALY, J. C. F. J. H. L. D. C. W. K. L. A. M. Flipping bits in memory withoutaccessing them: an experimental study ofdram disturbance errors: Acm sigarchcomputer architecture news: Vol 42, no3. *ACM SIGARCH ComputerArchitecture News* (2014).