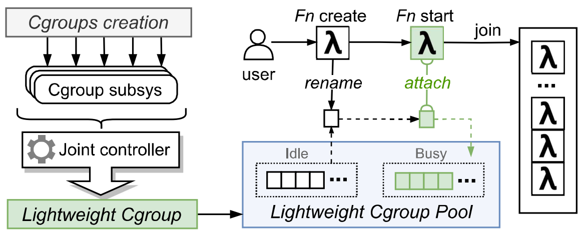


Figure 9: The lightweight Cgroup aggregates all subsystems, eliminating the time-consuming creation by renaming from the Cgroup pool.

(aka the *cpu*, *cpuacct*, *cpuset*, *memory*, and *blkio*) into one single dedicated lightweight cgroup. The implementation of the joint cgroup controller helps RunD reduce the redundant cgroup operations when a container is started, significantly decreasing the total number of cgroups and system calls.

The cgroup pool with renaming mechanism eliminates the time-consuming cgroup creation and initialization. RunD pre-creates corresponding lightweight cgroups and maintains them in a cgroup pool based on the pre-defined node capacity. These cgroups are marked idle when initialized, and are protected in a linked list. For each created container, RunD simply allocates an idle cgroup, updates the state to busy, performs the `cgroup rename` operation, and then attaches the container to this renamed cgroup when a container is started. If a container triggers recycling, RunD will take the cgroup back to the pool, kill the corresponding instance process, and then update the returned cgroup state to idle for subsequent allocating and renaming.

Adopting the above optimizations in kernel mode, we replay the evaluation in Section 3.3. The cgroups creation only consumes $0.09s$ (1 thread), $0.1s$ (50 threads), and $0.14s$ (200 threads), respectively. Compared with the default mechanism, the lightweight cgroup and the rename-based cgroup pool reduce 94% of the cgroups creation time.

5 Evaluation

In this section, we evaluate the performance of RunD in supporting high-concurrency startup and high-density deployment of secure containers, and introduce the performance of RunD in production usage.

5.1 Evaluation Setup

We have implemented and open-sourced RunD with Rust, a more memory-efficient and thread-safe programming language. RunD runtime involves four main modules: Containerd-shim (21k LOC), Device (4.4k LOC), Hypervisor (5.6k LOC), and Lightweight-cgroup (20k LOC).

Table 1: Experiment setup in our evaluation.

	Configuration	
Hardware	CPU: 104 vCPUs (Intel Xeon Platinum 8269CY) Memory: 384GB, two SSD drives: 100GB, 500GB	
Software	OS: CentOS7, kernel: Linux kernel 4.19.91	
Container	kata-qemu	containerd 1.3.10, kata 1.12.1
	kata-FC	containerd 1.5.8, kata 2.2.3
	kata-template	containerd 1.3.10, kata 1.12.1
	RunD	containerd 1.3.10

Baselines: we compare RunD with the state-of-the-art secure container, Kata Containers [19]. Specifically, we use three popular configurations of Kata containers: *Kata-qemu*, *Kata-template*, and *Kata-FC*. *Kata-qemu* uses QEMU [15, 23] as the microVM hypervisor, *Kata-template* uses QEMU while integrating container template, *Kata-FC* uses lightweight FireCracker [20] as the microVM hypervisor. Kata-qemu and kata-template use an old version of Kata Containers, as the new version has some bugs that result in poor performance. Table 1 shows the detailed setups.

Testbed: we run the experiments on a node with 104 virtual cores, 384GB memory, and two SSD drives of 100GB and 500GB. Such specification is widely-used in production clouds. The 100GB drive is used as the root filesystem of the host operating system, and the 500GB drive is used by the secure containers. We use Alibaba Cloud Linux 2 for RunD and Alpine Linux [3] for others, as the guest operating systems in the microVM for a low memory footprint.

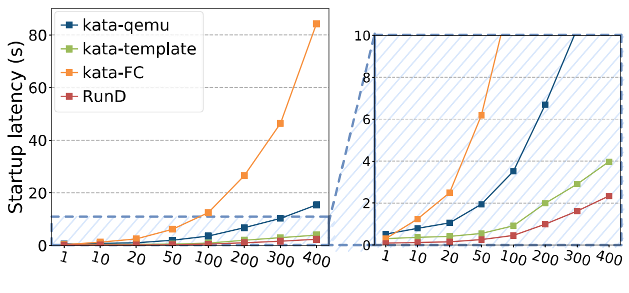
Measurement: in the CRI specification [6], a pod sandbox refers to a microVM with a lightweight pause container [12]. In all the tests, we only create the pod sandboxes without other containers inside, through the `crictl` command. In the following evaluations, the memory specification of a container denotes the size of memory that can be used by itself. The actual memory usage of a container is collected using the `smem` command.

As RunD is proposed to maximize the supported container startup concurrency and deployment density, in the experiment, we start empty secure containers without user codes or data considering that it is a common practice in FaaS to start empty containers concurrently for prewarming. The in-production results show the performance of RunD for actual workloads with all the steps involved.

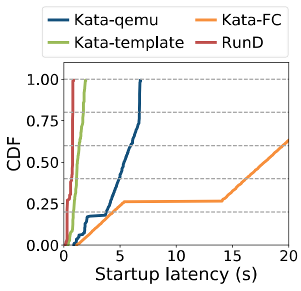
5.2 Concurrent Startup Measurement

In this experiment, we focus on three critical metrics related to user experience: *(1) the time needed to start a large number of sandboxes concurrently*, *(2) the startup latency distribution of the sandboxes*, and *(3) the CPU overhead on the host*. The first metric reveals the throughput of starting sandboxes, and the second metric reveals the experience of every user.

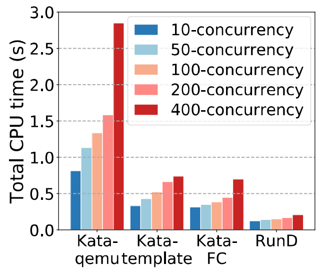
As for the first metric, Figure 10(a) shows the time needed to start a large number of sandboxes concurrently. In the fig-



(a) End-to-end startup latency with different concurrency



(b) Latency distribution



(c) CPU time

Figure 10: The startup metrics with different runtime and concurrency: (a) The end-to-end latency of concurrent startups. The right figure is an enlargement of the left one ($y \in [0, 10]$). (b) The CDF of startup latencies from a 200-way concurrent launch. (c) The CPU usage of concurrent startups.

ure, the x-axis shows the number of sandboxes to be started concurrently, the y-axis shows the overall time needed to startup all the sandboxes.

As shown in the figure, RunD uses the shortest time to start a large number of sandboxes for all concurrency levels. When 200 containers are created concurrently (we already observe such high-concurrency in Alibaba serverless platform), Kata-FC, kata-qemu, kata-template, and RunD needs 47.6s, 6.85s and 2.98s and 1s to create them. Kata-FC requires a much longer time to startup the sandboxes when the concurrency is high. This is because Kata-FC uses *virtio-blk* to create *rootfs*, and the performance is poor at high-concurrency, as we measured in Section 3. There is no such bottleneck in Kata-template and Kata-qemu. Kata-template simply uses template to reduce the overhead of guest kernel and *rootfs* loading, but the inefficient *rootfs* mapping, code self-modification and high host-side overhead of the cgroup operations still exists. As a result, it performs worse than RunD at high startup concurrency. The overall optimizations suggest that RunD provides the performance improvement of about 40% over its nearest baseline, Kata-template, at high-concurrency (e.g., 400-way) startup.

As for the second metric, Figure 10(b) shows the latency distribution of starting each sandbox, when 200 sandboxes are started concurrently. RunD and Kata-template are able to start sandboxes in a stable short time, but the latencies of starting sandboxes with others are out of expected. Users can have identical good experiences with RunD.

As for the CPU overhead, Figure 10(c) shows the CPU time needed on the host to startup sandboxes. When the concurrency is high, RunD greatly reduces the CPU overhead. For instance, when 200 sandboxes are started concurrently, RunD reduces 89.3%, 74.5% and 62.1% CPU overhead compared with Kata-qemu, Kata-template, and Kata-FC, respectively. In addition, the CPU overhead of RunD only increases slightly, when the concurrency increases. This is due to the read/write split policy and the reduction of compute-intensive operations in cgroups. Therefore, RunD is scalable in starting more sandboxes concurrently.

*In summary, RunD is able to start a single sandbox in **88ms** and launch **200** sandboxes simultaneously within **1s**, with minor latency fluctuation and CPU overhead.*

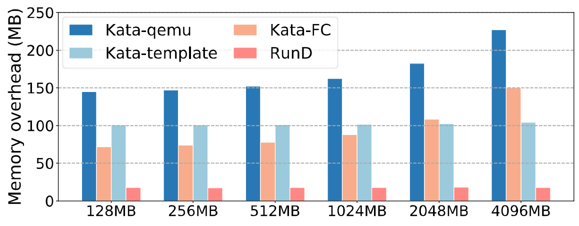


Figure 11: The memory overhead of Kata-qemu, Kata-template, Kata-FC, and RunD (100 sandboxes are deployed).

5.3 Deployment Density

In this experiment, we evaluate the effectiveness of RunD in increasing the sandbox deployment density. In general, the memory used by each container determines the deployment density, while the CPU time needed by each function invocation is minor in the serverless platform. Figure 11 shows the memory overhead when 100 sandboxes are deployed on the experimental node. In the figure, the x-axis shows the memory specification of each sandbox.

As observed, RunD has the least memory overhead among four runtimes, and does not increase with the memory specification. The memory overhead is less than 20MB per sandbox with RunD. Compared to kata-qemu, kata-template and kata-FC, the overhead of RunD is reduced by 54.9%, 27.2%, and 18.9%, respectively, even when the memory specification is 128MB. The memory overhead does not increase, because the microVM template technique uses the on-demand memory loading for the containers. Therefore, the page table required for memory management is determined by the actually used memory space. On the contrary, the memory overheads introduced by Kata-qemu and Kata-FC increase

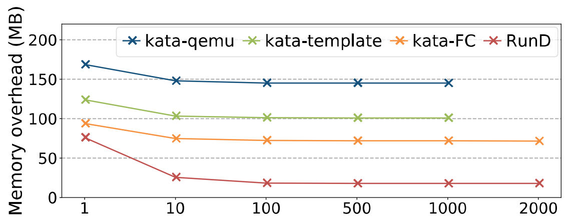


Figure 12: The memory overhead and the amortization by multiple secure containers. The missing point around 2,000 indicates the over-subscription for physical memory space.

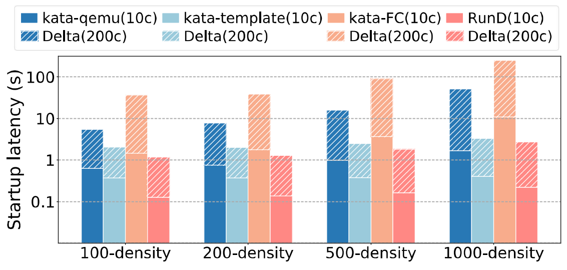


Figure 13: The end-to-end startup latency at different deployment densities. (10c/200c means a 10/200-way concurrent startup, and the Delta means the overhead increment compared with a 10-way concurrent statrup).

with larger memory specifications, as the page table is built for all available memory. In addition, the pre-patched kernel image in RunD further reduces memory overhead.

Figure 12 shows the average memory overhead of the sandboxes when different numbers of sandboxes are deployed on a node. The x-axis shows the deployment density. As observed, the average memory overhead reduces with the deployment density, as the sandboxes share the mapped code/data segments. RunD reduces the memory overhead by 87.7%, 82.4%, and 75.1% when 1,000 sandboxes are deployed, respectively, compared with kata-qemu, kata-template, and kata-FC.

RunD supports to deploy over ***2,500*** *sandboxes of 128MB memory specification on the node with 384GB memory.*

5.4 Impact of Deployment Density on Startup Latency and Concurrency

When some sandboxes are already deployed on a node, the performance of starting sandboxes concurrently is affected. Figure 13 shows the time needed to boot 10 and 200 sandboxes, when some sandboxes are already deployed on the node. The x-axis shows the number of already deployed sandboxes. The y-axis is in the log10 scale.

When 1,000 sandboxes are already deployed, the time needed to startup 10 containers increases by 1.69s, 0.41s,

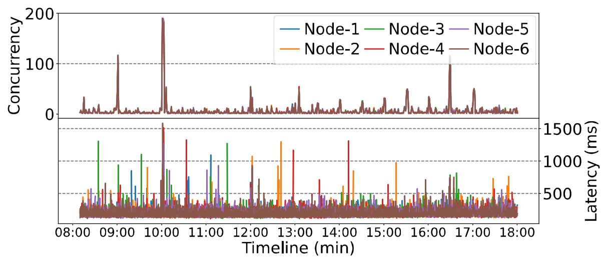


Figure 14: The startup latency and concurrency tracing of RunD in Alibaba serverless platform.

10.8s, and 0.22s compared with the cases in Figure 10(a) with Kata-qemu, Kata-template, Kata-FC, and RunD. In addition, the time needed already increases with the number of already deployed sandboxes.

We can also observe that, the time needed to start 200 sandboxes is at least 10 times as much as that needed to start 10 sandboxes at a 1,000-density deployment in all the tests. The significant increase originates from a large number of cgroups in the host operating system. Scheduling and managing containers with these cgroups consume more CPU cycles, thus resulting in CPU bottlenecks appearing earlier than a low-density deployment. The increased time is the smallest with RunD, because it already eliminates many time-consuming cgroup operations.

RunD shows better performance and stability in supporting high-concurrency startups at high-density deployment.

5.5 In-Production Usage for Serverless

Currently, Alibaba serverless computing platform has adopted RunD. The platform serves almost 4 billion invocations from more than 1 million different functions per day.

Figure 14 reports the sandbox startup concurrency and the corresponding startup latency from six nodes. The specification of each node is the same as our experimental setup in Table 1. The data is collected between 08:00 and 18:00 of Jan 10^{th}, 2022. There are about 800 active sandboxes on each node, when the concurrency data is collected. The in-production startup latency of sandboxes at high-concurrency is consistent with that reported in Section 5.4.

As observed from the figure, the startup concurrency bursts at the beginning of each hour. At most 191 sandboxes are started concurrently around 10:00. RunD starts the 191 sandboxes in 1.6 seconds. We look into the function invocation logs, and find that the periodic burst is caused by the an-hour time trigger and cluster-level load balancing. The periodical burst is pervasive, as the Azure serverless platform traces [14] show the same pattern. In the figure, the sandbox startup latency occasionally increases when the concurrency is low. The long time results from the operation in loading large-scale workloads from the tenants. Although the startup

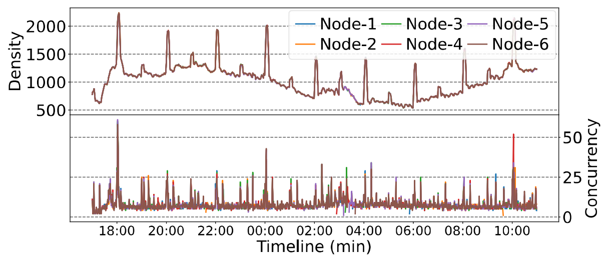


Figure 15: The deployment density and concurrency of RunD in 1 minute intervals, from Alibaba production traces.

concurrency is not always high, it is crucial to ensure a quick startup for a good user experience.

Figure 15 shows the deployment density of the sandboxes on each node. We collect the density statistics of the six nodes between 18:00 of Jan 10th to 10:00 of Jan 11th, 2022. As observed, more than 2,000 sandboxes are deployed on a node at most. We can also find that the high-density deployment happens at the same time as high concurrent startups. This is because many tasks are triggered at the beginning of each hour. The deployment density does not achieve the theoretical upper limit of $384GB/(128+20)MB$=2656 containers, as some functions use more than 128MB memory, and the workloads are also balanced to other nodes.

RunD is production-verified to meet the high-concurrency startup and high-density deploymant requirements.

5.6 Lessons Learned from Production Usage

Besides the RunD secure container, we have some insights about designing secure containers for serverless systems.

Lesson-1: the CRI specification designed for Kubernetes is not suitable for serverless system. In CRI, multiple related containers can co-locate in the same sandbox, and a lightweight pause container is started first to prepare the cgroups for the remaining containers. This pause container-based solution is negative for serverless computing, as each sandbox only has a single container for security and privacy.

Lesson-2: Functions tend to use the same standard guest environment provided by the serverless platform. In this case, the language environment (e.g., JVM) can also be integrated into the microVM template. However, the language-level template will invalidate the on-demand memory loading in the guest because some language runtimes need to pre-allocate the available memory. There are tradeoffs between higher memory utilization and less startup time, and the decision should be made based on how often the functions share the language environment.

Lesson-3: The memory usage of user functions is the key aspect determining the upper limit of the deployment density. Most functions are lightweight. When 2,000 sandboxes are deployed on a node of our serverless platform, the CPU utilization does not achieve 50%, and there is no complaint on the poor performance from users. One reason for the low CPU utilization is that many sandboxes are actually idle and “kept-alive” after its function invocation is completed in a serverless scenario.

6 Related Work

The most closely related work to RunD is FireCracker [20], which proposes a lightweight VMM for serverless runtime. It provides fast startup within 125ms, allowing 150 VMs to start concurrently per second per node, with less than 5MB footprint per VM. However, FireCracker only serves as the hypervisor stack in the Security Container model, without other complex related processes, e.g., *rootfs* [52]. By contrast, RunD investigates the guest-to-host solution through all stacks and provides higher concurrency and density.

Higher-density deployment. Regarding serverless computing, in the space of higher function deployment density of Secure Containers and VMs [57], the key is designing a more lightweight container runtime both in guest and host. Unikernel [36, 37, 43, 47] runs as a built-in GuestOS without necessary add-ons, demonstrating great potential for deploying containers with less overhead. Kuo [33] Explores lightweight guest kernel configurations for use in Unikernel environments, which has similarity to the approach towards reducing guest kernel size. However, Unikernel is hard to be changed once after compilation with the application. Its compile-time invariance results in poor flexibility in practice. SAND [21] adopts the *multi-container-per-VM* model to amortize the memory footprint of sandboxing. However, they do not further investigate the utilization impact of memory fragmentations in a real-system with high-density deployment. Gsight [61] observes that fine-grained function-level profiling can expose more predictability system-level features in the partial interference. With a more accurate interference predicting [27, 44], the function density can get improved with QoS guaranteed.

The above studies make sense in improving the effective density with less interference for serverless. They are orthogonal to our work, because RunD is motivated to improve the maximum deployment density on a signe node.

Higher-concurrency startup. In the space of higher function startup concurrency, recent approaches leverage the container prewarm pool [9, 40, 49, 58]. The state-of-the-art on container prewarming, SOCK [42], uses a benefit-to-cost model to select packages pre-installed in zygotes, and builds a tree cache to ensure that the forked zygote container does not import any additional packages other than the private ones the handler specifies. The C/R (Checkpoint/Restore) [7, 31, 39] supporting the VM snapshotting [10, 28, 29, 41, 54] captures the state of a running instance as a checkpoint, and then restores it once cold startup. Observing that most functions only access a small fraction of the files and mem-

crictl command to start pod sandboxes and measure the time between the first *crictl runp* invocation and the last ready pod sandbox. For measuring the memory footprint, we use the *smem* command and the PSS column of its output. All the tests are run on a machine with 104 vCPUs and 384GB of memory running CentOS7.

A.2 Artifact Check-list (Meta-information)

- **Run-time environment:** Alibaba ECS instance;
- **Hardware:** Intel Xeon(Cascade Lake) Platinum 8269CY, CPU and Memory: 104 cores and 384GiB, Storage: Two ESSDs (100GB + 500GB);
- **Software:** Aliyun Cloud OS 2, with Linux kernel 4.19.91, Kata container 1.12.1 and 2.2.3, containerd 1.3.10, smem 1.4;
- **Metrics:** average latency and average memory footprint;
- **Time is needed to complete experiments:** 10 hours;
- **Available:** https://github.com/chengjiagan/RunD_ATC22
- **Code Licenses:** Apache-2.0 license

A.3 How to Access and Installation

Github Link: https://github.com/chengjiagan/RunD_ATC22. Then you should follow the *README* instructions to get installation.

A.4 Experiment Workflow

A.4.1 High-concurrency Experiment (Section 5.2)

Scripts are provided to run the high-concurrency test for kata-qemu, kata-fc and kata-template. To run high-concurrency tests:

```
$ ./script/time_kata_test.sh
$ ./script/time_katafc_test.sh
$ ./script/time_katatemplate_test.sh
```

They may take several hours to finish. Some concurrency tests can be removed by removing the corresponding concurrency setting in file *time_test.conf* to shorten the time. The scripts will create a directory (e.g., named like *time_kata_05120948*) to store the logs.

We provide python scripts to analyze logs from the tests:

```
$ python3 data/time.py
$ python3 data/cpu.py
```

The python script will create two .csv files in the result directory: *time.csv* and *cpu.csv*. Each line in the csv file indicates the average cold-start latency and cpu time of a container runtime.

A.4.2 High-density Experiment (Section 5.3)

Scripts are provided to run the high-density test for kata-qemu, kata-fc and kata-template. To run high-density tests:

```
$ ./script/mem_kata_test.sh
$ ./script/mem_katafc_test.sh
$ ./script/mem_katatemplate_test.sh
```

Density and memory capacity of containers in the tests can be changed in the file *mem_test.conf*. The scripts will create a directory named like *mem_kata_05120948* to store the logs.

We provide a python script to analyze logs from the tests:

```
$ python3 data/mem.py
```

The python script will create a csv file for each runtime, named like `mem_kata.csv`, containing the average memory consumption of containers with different memory capacity in different density.

A.4.3 Density Impact on Concurrency (Section 5.4)

Scripts are provided to run the high-density test for kata-qemu, kata-fc and kata-template. To run tests:

```
$ ./script/density_kata_test.sh
$ ./script/density_katafc_test.sh
$ ./script/density_katatemplate_test.sh
```

The background density and the concurrency of the tests can be changed in the file *density_test.conf*. The scripts will create a directory (e.g., named like *density_kata_05120948*) to store the logs.

We provide a python script to analyze logs from the tests:

```
$ python3 data/density.py
```

The python script will create a csv file for each runtime, named like *density_kata.csv*, containing the average cold-start latency under different background densities and concurrencies.

A.5 Expected Results and Notes

The expect results are all stored in *ae_data* directory. Considering that some related binary packages are tightly integrated with our internal system, we provide a screencast *ATC_RunD_AE.mp4* of the tool along with the results. You can also find RunD-related performance and execution logs in our artifact..