Dockerust : Rust-based Container Runtime

Youhan Wu, Zhaocheng Huang
Rice University

1 Introduction

Container-based virtualization is really popular in recent years. Thanks to the advent of Docker, container technology has subverted the traditional systems development life cycle and refactored the way people build and deploy software. While Docker is considered more lightweight than other virtualization technologies, it is still not efficient enough in a high load environment, as well as in a resource limited platform such as IoT devices [1], with high overhead incurred by Go runtime.

Docker introduces an underlying container engine to help start, terminate processes and multiplex system resources, which utilizes Linux namespaces [2], as well as a lot of system ABIs, such as read/write, exec, fork, etc.. However, many existing container runtimes like runc, gVisor are written in Go, which introduces overhead to Linux system call and resources utilization [3] [4]. From this perspective of view, Go might not be the best programming language for this task. It doesn't have a complete support for the fork/exec model of computing. Besides, the container sill suffer from a performance cost due to language runtime. There do exist some alternative container runtime written in system languages with relatively low overhead. In 2017, Red Hat introduced crun to replace runc. However, C is not recognized as a memory safe language, it can expose too many vulnerabilities due to its inherent defect. Memory issues account for ??%(TODO) of crun-related CVEs. One simple memory-related vulnerability may cause huge economic losses to container users.

We want to implement an OCI-compliant runtime in Rust to explore the potential of being faster, safer and more resource-friendly. Compared with other popular runtime engines, Dockerust is faster than runc due to its built-in binding for raw system calls, and requires fewer system resources without needing of a GC or thread schedule. Morever, it is much safer than crun with restrict compile-time type checking and data race detection. To sum up, Dockerust strikes the balance between performance, security and cost, which makes it the perfect fit for general-purpose scenarios, and has an edge in resource-limited platforms, such as single-board computers.

We will perform a benchmark analysis of the above mentioned runtimes, and provide a comprehensive report from the performance, resource usage, and security aspects. And to further confirm our conclusions, we will create and start an accessible nginx server image respectively, and run a load test against the server to see under the same circumstances, how many QPS the server can handle.

The anticipated output includes a runnable OCI-compliant runtime, a set of benchmark testing programs, and a final report.

2 Background

Docker is one of the most popular technologies for containerization. Instead of boosting the whole system environment in VMs, Docker utilizes the Linux kernel's resource isolation and constraint features to share a kernel across containers. Docker can create isolated environment for containers in different levels (e.g., network, PIDs, UIDs, IPC) with support of Linux namespaces [5]. Besides, it can control resource allocation and limitation among containers using cgroups.

Rust is a programming language that emphasizes performance and memory safety. Rust aims to be as efficient as C++. It doesn't perform garbage collection, and it has features like zero-cost abstractions and inline expansion to optimize the compiled code. Rust is designed to be memory safe. It does not permit null

pointers, dangling pointers, or data races, and safety and validity of the underlying pointers is checked at compile time. Rust has been growing fast in recent years, and has been widely used in many large projects. It is reported that Rust is incorporated into the forthcoming 6.1 Linux kernel, which is an important milestone for Rust, and also a good signal for our project.

We believe with official support from system level, and the brilliant features from the language level, a new container runtime written in Rust will be a better alternative to existing runtimes.

3 Methods and Plans

In order to build a OCI-compliant runtime, we need to study the Runtime Specification first to see what a minimal runtime should include. So that we can swap the Docker's runtime from runc to Dockerust, and then furthermore perform a benchmark testing against other runtime implementations.

Apart from the specs, we would dive into the Docker's source code, to see how exactly a container is created and started, and how the system resources are isolated and allocated. We will try to port the core components to our Rust implementation.

If everything can be delivered on time, we will perform a benchmark testing over the some of popular implementations and Dockerust. The expected evaluation criteria includes performance (e.g. wall time for container startup), resource utilization (e.g. memory usage, cpu usage), and security [6] (e.g. fuzzing, perform an attack exploiting CVE samples).

3.1 create a namespaces

What we talk about when we talk about containers? In Linux context, a container is just a forked process. It's nothing fancy about it, expect it is running in an isolated namespace.

Namespaces are a feature of the Linux kernel that provide the logical isolation of resources for processes running in different sets of namespaces. There are different types of namespaces, including IPC, Network, Mount, PID, User, UTS, and so on.

Linux provides the *CLONE(2)* syscall to fork a child process and allow it to create or enter new namespaces, as long as the namespace flags are specified.

When we run the command

```
dockerust run /bin/ls
```

,internally, it will be translated as a new command

```
dockerust init /bin/ls
```

. This commnad is running in a new execution context via the *clone* syscall. If we print out the process tree at this point, we will see the init process and the only process in the namespace is this command. Then we will run the syscall *EXECV(3)* to replace init process to */bin/ls*. Now we can say we have successfully jailed the container process. But that's not enough.

3.1.1 Virtual File System

3.1.2 Virtual Network

3.2 limit and monitor system resources

3.3 IPC with pipes (or sockets? TBD)

4 Protections

The Dockerust is running in privileged mode, so extra protections must be applied to mitigate privilege escalation attacks.

Here are a bunch of protections we have used:

4.1 Seal the executing program

A malicious container can overwrite the dockerust runtime binary and thus gain root-level code execution on the host. Examples are CVE-2019-5736, ...(TODO)

When the user image is run, it must be the init process with PID equals 1 in ths procfs and with all the constraints ready. In our implementation, when the user enters *dockerust run <image>*, dockerust creates a *dockerust init <image>* subprocess in a anmespace that is already set up. Then, dockerust init the image, by calling *execv* syscall to overite itself with the user-provided image binary.

However, if an attacker pass */proc/self/exe* as the image url to *dockerust run <image>* command, the attacker will have root access to the */proc/[dockerust-pid]*. By using it as a reference to the dockerust binary and overwrite it, the attacker can achieve code execution on the host as root.

methods: using memfd syscall to create an anonymous file fd for */proc/self/exe* TODO

5 Evaluation and Results

To explore the potential of Rust being faster and memory saving, we measured the running time and memory usage of two programs written in Rust and Go that perform the same system call. We decided to measure the performance of system call because our container runtime relies on system calls to complete most of the operations to run a container, like mounting a storage directory or forking a process. We choose Getppid because it is simple and it makes us easier to test the latency between the program and Linux ABI.

We wrote two small programs in Rust and Go and performs Getppid system call for 100,000,000 times. The Rust program uses assembly instructions, and the Go program uses internal syscall library. Both are the best way we know about, and the Go syscall library is used by runc as well when initializing and running a container. Then we tested the two programs on a laptop with AMD Ryzen 7 5800H processor and 16GB RAM running Fedora Linux 36. We compiled the two programs on the laptop using official compilers. We allowed default optimization of the Go compiler and set the build target of the Rust compiler to release for best performance. We tested each program 10 times and recorded the running time in each run. We also recorded the memory usage in a separate run using Valgrind. The result is shown in Table 1 and 2 and Figure 1.

The result shows that the average running time of the Rust program is 8.45% less than the Go program, and the average memory usage of the Rust program is 5.13% less than the Go program. Therefore, we believe Rust has the potential of being faster and saving more memory when executing tasks that require a lot of system calls, and we believe Rust will be suitable for our container runtime.

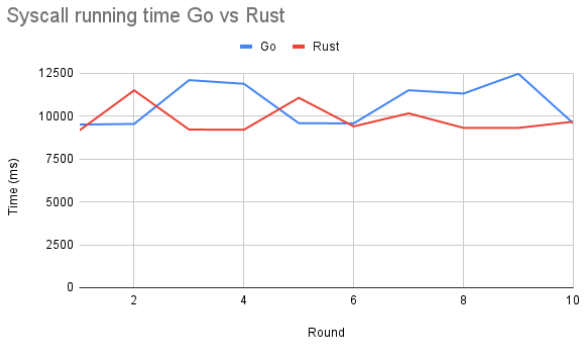

Figure 1: Running time of the two demo programs.

Memory Usage (B)	Average	Max	Min
Rust	2960.096386	7656	312
Go	3120.177778	62944	2448

Table 1: Memory usage of the two demo programs.

Running time (ms)	Average	Max	Min
Rust	9802.7	11502	9162
Go	10706.9	12469	9507

Table 2: Running time of the two demo programs.

6 Discussion and Lessons Learned

In the evaluation process, we found that the Rust program is smaller in memory usage and faster. We believe the reason is that Rust does not have a runtime like Go, and it reduces the memory usage and latency when doing system calls. We have completed the first part of our project which allows creating isolated namespaces using Linux kernel features and run a process within the namespaces. And we believe we can utilize the efficiency of Rust language in our final product.

7 Conclusion

We plan to build a container runtime based on Rust. Our evaluation on Rust and Go runtime shows that Rust has the potential to run faster and save memory. We have concluded the main components we need to implement in a OCI-compliant runtime. In the future, we plan to deliver the product and perform benchmarks from performance, resource usage, and security aspects. We expect the anticipated output will maintain a balance between performance, resource usage and security.

8 Milestones

10/13 Project Proposal

10/16 Study OCI runtime-spec

10/23 Study Docker Architecture

10/30 Warm up Rust lang

11/01 Project Midterm Report

11/06 Port part of core components

11/08 Project Midterm Meetings

11/13 Port the rest of components

11/20 Write benchmark testing programs

12/01 Project Presentations

References

[1] Antonio Celesti, Davide Mulfari, Maria Fazio, Massimo Villari, and Antonio Puliafito. Exploring container virtualization in iot clouds. In *2016 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 1–6, 2016.

[2] Karl Matthias and Sean P Kane. *Docker: Up & Running: Shipping Reliable Containers in Production*. " O'Reilly Media, Inc.", 2015.

[3] Ethan G Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. The true cost of containing: A {gVisor} case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*, 2019.

[4] Emiliano Casalicchio and Vanessa Perciballi. Measuring docker performance: What a mess!!! In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*, pages 11–16, 2017.

[5] Babak Bashari Rad, Harrison John Bhatti, and Mohammad Ahmadi. An introduction to docker and analysis of its performance. *International Journal of Computer Science and Network Security (IJCSNS)*, 17(3):228, 2017.

[6] Sari Sultan, Imtiaz Ahmad, and Tassos Dimitriou. Container security: Issues, challenges, and the road ahead. *IEEE Access*, 7:52976–52996, 2019.