Bachelor Degree Project

A COMPARISON BETWEEN NATIVE AND SECURE RUNTIMES

Using Podman to compare crun and Kata Containers

Bachelor Degree Project in Information Technology
22,5 HP
Spring term 2021
Course IT610G

Fredrik Björklund
b18frebj@student.his.se

Supervisor: Johan Zaxmy
Examiner: Jianguo Ding

Table of contents

Abstract: Containers is a widely used way of developing and delivering software today. As they take use of abilities in the kernel to provide isolation and control, they provide a small overhead compared to traditional Virtual Machines. But with using a shared kernel comes additional security threats. A solution to this is to provide a extra layer of virtualization to provide extra isolation.

The aim of this research is to study two different runtimes. The selected runtimes are Crun and Kata Containers. Where as Crun is a native low-level runtime and Kata Containers offers an additional layer of isolation.

To test these runtimes, this study use a Python benchmarking suite called pyperformance, to be able to measure what modules and libraries are affected by this extra layer of isolation.

The findings are that the overhead in ranges from <1x up to 44x comparing the two runtimes. This research show what modules and libraries in Python are affected in a significant way when executed in Kata Containers.

Keywords: Containers, Crun, Kata Containers, Python, Virtualization, Security, Benchmarking.

1 Introduction

Since the early 2010s containers have changed IT operations in business. Containers are a light-weight way of launching and hosting services, often known as microservices. They are an easy way to package the (bare) necessities, such as code, runtime, system tools, libraries, and settings (Docker, 2021), to run an application. Containers take use of kernel modules to achieve this, but with sharing a kernel amongst different services comes a security risk. To negate this security risk, one solution is to create a lightweight virtual machine isolating the kernel between services, and with this added layer of abstraction comes a performance overhead.

This study will compare running code in a lightweight vs isolated runtime. The code selected for the study is Python. Chapter 2 will cover background needed for the reader to understand the concepts of containers and runtimes. Chapter 3 will focus on related work and what have been done in the field hitherto. Chapter 4 defines what this research contributes. Chapter 5 and 6 will introduce the reader to the steps taken to conduct and apply method and experiment setup to aim to solve the selected problem. Lastly this is followed by an analysis of the collected data and to end with a discussion of the result.

2 Background

This section will cover some technical part that is necessary for further reading. The topics discussed are the basic foundation of containers, runtimes and Python. When containers are discussed, it mainly refers to Linux containers. First the reader will be introduced to some scenarios where containers are used, this is followed by an explantion of the underlying technologies that makes these use-cases possible.

2.1 Containers

Containers are a portable, lightweight and have an easy-to-manage lifecycle (McGee, 2021) it has become an alternative to the Virtual Machine (VM). Containers have been widely adopted and associated with cloud infrastructure and operation as *Infrastructure-as-a-Service* and *Platform-as-a-Service* (Casalicchio & Iannucci, 2020). Furthermore, it has been seen in scientific communities to provide a richer learning experience for students giving them more a hands-on approach to learning (Mehringer & Barker, 2020). Zheng and Thain (2015) provide another example of the wide range of variety that containers have been used in, where the authors use containers to adopt *Workflows* - a segment of tasks in large scientific applications - in a containered environment.

Containers consist of three combined features in the Linux Kernel: Namespaces, control group (cgroup) (Kernel, 2021**)** and capabilities.

Namespaces create an abstraction for global system resources and from the point of view of a process located in that namespace it looks like it got its own isolated instance. (“namespaces(7), Linux manual page”, 2021) Namespaces take use of three system calls. *clone()* which creates a new process and namespace where the process is attached to the new namespace, *unshare()* that only attaches a process to a newly created namespace and, *setns()* makes it possible for a process to join an existing namespace. The effect of this is that the process(es) inside the namespace appears to have its own operating system, while in reality it shares kernel with other namespaces. A Linux system can create namespaces for *cgroup*, *Inter-Process Communication* (IPC), *network*, *mount* points, *process ID* (PID), *time* (Boot and monotonic clocks), *user* (and group IDs) and *UTS* (Hostname and NIS domain name).

cgroup allows the operating system to hierarchically organize processes and distribute resources. It mainly consists of two parts, the core, and controllers. The core is responsible for the hierarchically organizing of

processes, while a controller's main function is to distribute a resource along the hierarchy. Control groups may be nested, i.e., children got the same limitation as their parents. That means that the limitation set to a container apply for the whole container.

Capabilities split the power of root into smaller pieces, which allows a program to only have the right capabilities to execute it purpose. For example; you may want a program to be able to *kill* but not to *chown*. This is to prevent a compromised programs to limit its potential damage. This is necessary for a container since it needs some capabilities as root without giving it more capabilities as needed. ("capabilities(7) - Linux manual page", 2021)

As mentioned, these features already exist in the Linux Kernel, they can be used by a (super)user in a manual way and be applied to the users need. Next section will cover how container managers use these three Kernel abilities to create and run containers.

2.2 Runtimes

A major part of the back end for running containers is the *runtime*. A runtime is the part of the container ecosystem that runs, as in execute and manage, container images. The container runtime can be seen as an automated process for applying the required features that was described in previous chapter.

In 2015 Docker, CoreOS and other container developers established the Open Container Initiative (Open Container Initiative, 2021) which is a collaboration to develop specifications to create compatible runtimes and container images. The result of this led to the creation of two specifications: (1) Runtime Specification (runtime-spec) and (2) Image Specification (image-spec). These runtimes are mostly modeled after *runc*, that is the runtime initially developed by Docker. *runc* was donated to the OCI-project "to be used as a cornerstone" for the project. Runtime-spec determines on how a runtime is constructed, i.e., what properties and type of data that can be passed to said property. This makes it possible for a user to switch runtime with a single command, if a different runtime is used there will be little to no rework for an organization's infrastructure and operation. This study will compare one native and one secure runtime

that is modeled after the OCI , *crun* and *Kata Containers*. Following is a brief explanation of the tools.

crun is similar to *runC,* as it is a native low-level runtime following the OCI runtime-spec. It is open-source and currently maintained by Red Hat as part of their larger containers eco-system (Containers, 2021). The main difference between runC and crun, is that runC is written in Go while crun in C. The perks of using C instead of Go, is a lower memory footprint and quicker start-up time. At the moment of writing, it is the default runtime for Red Hats container managing tool Podman.

kata-containers (Kata Containers, 2021) are an open-source merging of Intel Clear Containers with Hyper.sh RunV. It's supported by the major hardware architectures as AMD64, ARM, IBM p-series and x86_64. provide an additional layer of security. Kata utilizes hardware virtualization to isolate the sandbox environment, this includes a dedicated kernel, network isolation, I/O, and memory. As it is a nested virtualization it needs VT extension. With a dedicated kernel, malicious attacks stay in the dedicated kernel and can't enter any other container, as opposed to a shared kernel.

2.3 Python

For this study, the original implementation of Python, CPython will be used and when Python is mentioned, it refers to CPython if it is not referenced otherwise.

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics (Python Software Foundation, 2021). Its easy-to-read syntax makes it a popular language for beginners and in addition makes its code easy to maintain. Python’s wide variety of use has made it a popular programming language. According to recent survey done by Stack Overflow, Python is the fourth most used programming language used by professionals with a market share of ~41% (Stack Overflow, 2021). The range of use cases varies from Data Science (Cao, 2017), scientific computing (Amela, Ramon-Cortes, Ejarque, Conejero & Badia, 2017) and finance (Varela, Wehn, Desmettre & Korn, 2017).

3 Related work

This study mainly draws its inspiration from Minì and Giorgio (2020), Viktorsson et al. (2020) and Li et al. (2017). The two former did similar studies to this study, comparing container runtimes, and the latter compared benchmarks on native, container and VM.

Minì and Giorgio (2020) research shows that the difference in kata and crun and runc is in some cases negligible and in other cases and up to ~17x slower in some test. They also run test of *gVisor*, which is another secure sandbox.

The findings of Viktorsson et al. (2020) are that Kata container have around 60% the performance in the applications they test.

Li et al. (2017) comparison between hypervisor and container shows a variety in results in their benchmark of CPU in containers, the result varies from feature-to-feature and job-to-job. This makes it interesting for researching on how real-world benchmarks differs in containers.

Other related work that should be mentioned is Yoshimura, Nakazawa & Chiba (2020) as they test different OS base images for containers to evaluate their performance for various tasks, such as web servers, caching and more. For testing base images for different Python versions, they use the pyperformance suite.

4 Problem Description

In this section we set out a scope and project focus, and explain the novelty of this research, what separates this research from previous mentioned research in the section covering related work.

4.1 Motivation

The technology covered in background, is commonly used for day-to-day operations in the IT industry. It is of interest to stress these technologies to find limitations or where they excel. Meanwhile there is a lack of research done in a similar manner as this study. Stressing different libraries and modules belonging to the Python programming language and see how they react to being in a secure runtime as Kata Containers contra native runtime as crun, can be beneficial in designing and using applications and produce results that can be used both by industry and researchers.

As mentioned in the introduction, the motivation for this experiment is to find modules and libraries in Python code running within different execution time in sandboxes. What a "good" execution time is subjective to the project or organization, but there is often a threshold of what is acceptable. Comparing how Python runs in different light weight vs isolated runtimes could be beneficial since the tests are mostly library-by-library and module-by-module, and this could give developers an understanding on how different libraries and modules perform in isolated runtimes.

4.2 Research Question

This study tries to answer the research question (RQ):

RQ: *"How does python code differ in terms of execution time in secure runtimes opposed to a default runtime?"*

If the research question is to be accepted, it will lead to explore a deeper exploration of which modules and libraries differ in execution time. This means that a second research question will be constructed.

RQ2: *"What modules and libraries differ?"*

This gives us two variables too measure: execution time and modules/libraries.

5 Method

The selected method for this study is to do an experiment. Doing experimentation is a suitable method for answering the research question. Since the study only measures wall time, the *why* can be difficult to explain since it will be more of research of a black box. With that said, the experiment will leave us with enough data and input for conducting further theorising and experiments if we will see different parts of python code that runs sub-optimally in a light weight VM.

To make sure the experiment has a well-defined foundation we will be using the template recommended by Wohlin et al. (2012).

This will ensure us that we have a goal to work towards. The workflow they recommend is following:

1. Scoping
2. Planning
3. Operation
4. Analysis and Interpretation
5. Presentation and Package

5.1 Scoping

The first step is to define the goal of the experiment. The importance of a scope is important since without a clear goal the following steps, planning, and execution, will be lacking a direction. To achieve a direction Wohlin et al. (2012) recommends a template (italics is authors own):

Analyze *Python code*

for the purpose of *evaluation*

with the respect to their *execution time*

from the point of view of the *developer*

in the context of *sandboxing*.

With the template completed, we know the "*why?"* of the experiment, we advance to the next step, planning, where the *"how?"* is answered.

5.2 Planning

The major step in an experiment is the planning. It is an important step because without a proper scheme, the likeliness to fail grows. This part consists of seven steps: Context selection, Hypothesis formulation, Variables selection, Selection of subjects, Choice of design type, Instrumentation, and Validity evaluation. The planning is influenced by the goal pin-pointed in chapter 5.1 and the following sections will shape the experiment design in chapter 6.

5.3 Hypothesis Formulation

This section covers the creation of two - a null and an alternate hypothesis. In later stages of the experiment, the data collection, we will see if the data reject the null hypothesis meaning that the alternate hypothesis is accepted. The alternate hypothesis is the *negation* of the null hypothesis and vice versa.

H0: There is no difference in *execution time* in *Python depending on the runtime.*

H1: The is difference in *execution time* in *Python depending on the runtime.*

5.4 Variable Selection

This section covers the selection of independent and dependent variables. Independent variables are the variables that can be controlled, and in this study that is the selection of software, such as programming language, runtimes, container engine. While the dependant is how we measure them, our metrics. The *execution time*.

The aim here is to explore how the independent effects the dependent.

5.5 Instrumentation

Instrumentation consists of three types, objects, guidelines, and measurement instruments.

Objects in this study is, Podman, crun, Kata Container and the source code for Pyperformance.

Measurement tools is *with* what tools we will collect data, this will be the benchmarking suite Pyperformance, to measure the execution time. Pyperformance is a collection of benchmarks ranging from a various selection of Python modules and libraries.

"The `pyperformance` project is intended to be an authoritative source of benchmarks for all Python implementations. The focus is on real-world benchmarks, rather than synthetic benchmarks, using whole applications when possible." ("The Python Performance Benchmark Suite — Python Performance Benchmark Suite 1.0.3 documentation", 2021)

Pyperformance can be seen in various tests in industry such as the Fedora project (Fedora Project Wiki, 2021), IBM (IBM, 2021) and in the speed.python.org project, a continuous testing of Python revisions.

There are some perks with running pyperformance. One major is that it makes it easy to benchmark seperate libraries to see how they perform within different runtimes. The benchmark is a mixture of both apps, games/puzzles, and simple testing of different libraries. It's also open source, which mean we can see how the benchmarks are executed from a code perspective, this can help with analysing the collected data.

The tool is open-source and available on GitHub. Before this tool is selected, the repository is checked for any currently known issues, and none are found in the repository that could affect this study.

Lastly, the guidelines are what will be discussed in the setup of the experiment.

6 Experiment Setup

This chapter covers the guidelines to get a reproducible result. The environment that the experiment is conducted in. This will cover topics like software and hardware versions.

6.1 Testing Environment

The experiment will be conducted in the laboration environment of NSA at University of Skövde. Between each benchmark, the system monitoring tool *top* will be used to see that the system reverts to an idle state, in addition, pyperf has a module *"compare_to"*, where the tests can be compared to see if the latest test doesn't show any oddities.

2 CPUs will be dedicated to the sandboxes, they will be the same set of units. CPUs for all the test, ensuring that they are isolated to the workload and produce stable results. In addition, 4 Gb of RAM is dedicated to the container.

The system that will be used for the test is composed as following:

Operating System	Ubuntu 20.04 LTS
Kernel	5.4.0-80-Generic
CPU	Intel i5-6600 4 CPU @ 3.9 GHz
RAM	32 GB
Podman	3.2.3
crun	0.20.1.15
Kata Containers	1.13
Python	3.9.5

The container manager selected is Podman. And the runtimes as previously mentioned is: crun and Kata Containers.

Next section will cover how the tests are grouped with a brief explanation of what they do.

6.2 Experiment overview

The execution of the tests is at large straight forward with no remarkable settings then the default. This section will cover an overview of the experiment.

The experiment will have Podman installed on bare metal, on Ubuntu 20.04. The first set of tests will be on crun, which comes installed with recent version of Podman. For this a containerfile (See Appendix A) is created which installs latest version of Python and the module Pyperformance. The Pyperformance benchmark will be ran with the rigorous flag to gain extra strength of validity. The rigorous setting conducts a larger number of tests per benchmark. For storing the results, a folder is created that is mounted on the container. The containerfile is then built with Podmans image building tool called “build”. After the image is built, it is run with Podmans run tool called “run”. This process is then repeated with Kata Containers. The tests run 20 times per runtime. Afterwards data will be extracted from the files and analysed in Excel on a separate system.

6.3 Tests

This section covers what type of Python executions categories that will be tested.

Apps – A selection of applications.

Math – Benchmarks mathematical operations.

Logging – Uses Pythons logging module.

SciMark – Selection of scientific computing benchmarks.

Serialize – Perform tests on JSON, XML and Pickle.

SQL – Mini-benchmark to test SQL.

SymPy – Test of the SymPy module.

Regex – Test Pythons ability to Regex.

Template – Test templating tool used with Python, such as Django and Mako.

Various – Tests that would not fit in a specific category, like games, puzzle, and start-up time.

7 Analysis of Data Collection

To conduct this analysis and get information on what the specific test do, the Pyperformance benchmarking documentation and Python3 documentation will be used. In the end of this chapter the reader will find a summary of all benchmarks and with calculated difference. To get a difference of either a single category, e.g., 7.1 Apps category, or full summary, the geometric mean of the tests done in the belonging runtime is calculated. This is of the recommendation of Fleming and Wallace (1986) paper *"How not to lie with statistics: the correct way to summarize benchmark results"*.

7.1 Apps

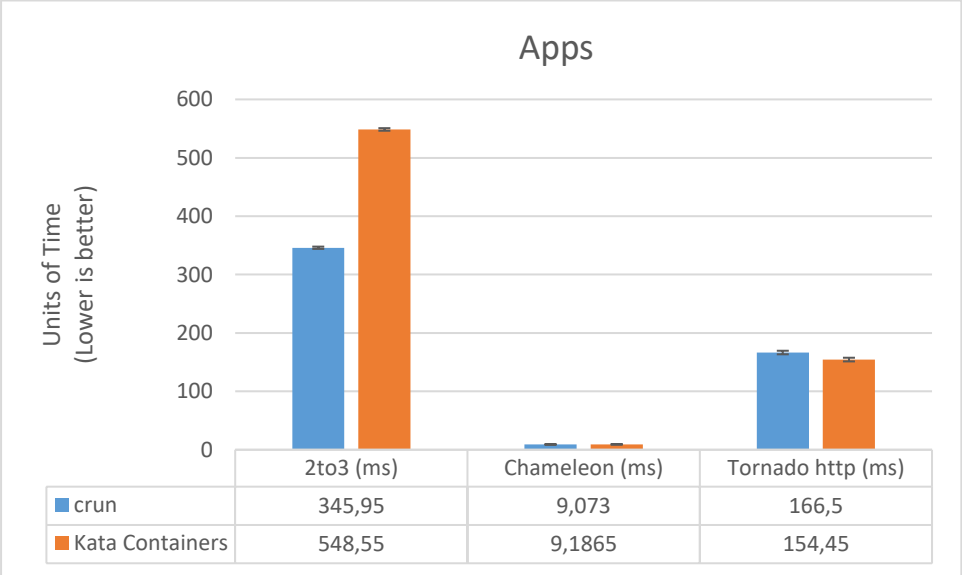


Figure 1 - Application performance

The first thing to note here is that Kata Containers performs *7.2% faster* than crun in the Tornado HTTP test. Though there is a *58.6%* performance regression in the 2to3 module. The 2to3 module is a translating tool that automatically translates code from Python version 2 to version 3. The total difference for this category is *14.2%* in favour of crun.

7.2 Math

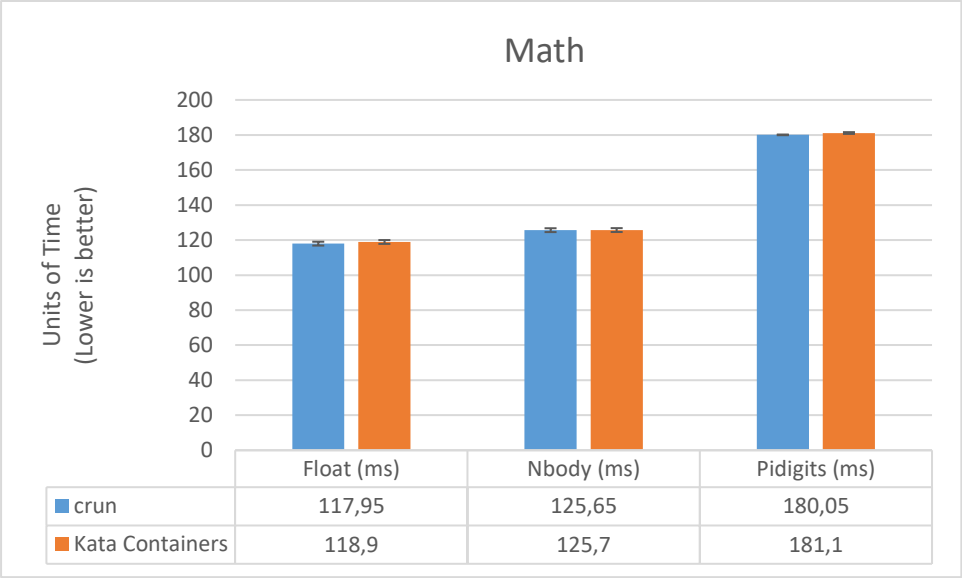


Figure 2 - Math performance

There is no notable difference in the pure math tests. These math test takes place mainly in the CPU. This confirms findings in other studies where pure CPU jobs doesn't come with a significant overhead compared to light weight runtimes. Both float and Nbody does floating point operations, while Pidigits calculates 2000 digits of pi, which stresses big integer arithmetic. All tests have a *<1%* difference and the total difference is *0.48%*.

7.3 Logging

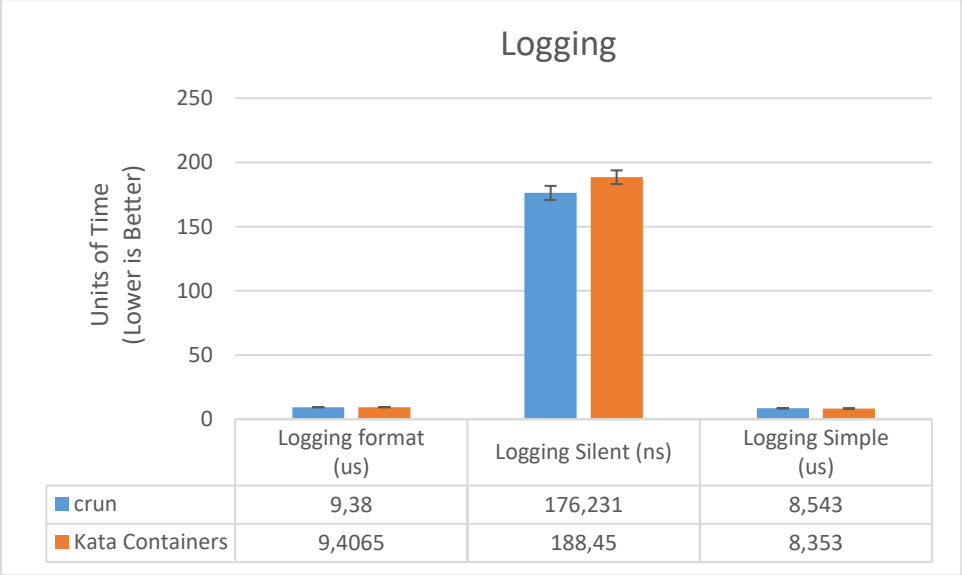


Figure 3 - Logging performance (note the different unit of time in the tests in parenthesis)

This module stresses Python's logging module by defining functions and classes to create a flexible event logging system for both applications and libraries. Format and simple test the *logging.warn*-function, while silent is a benchmark of debug.

Logging format have barely any overhead, only producing *0.28%*. Simple is *2.22%* faster in the test. Tough Logging Silent overhead is *6.93%*. This leaves the logging category with a *1.59%* total overhead.

7.4 SciMark

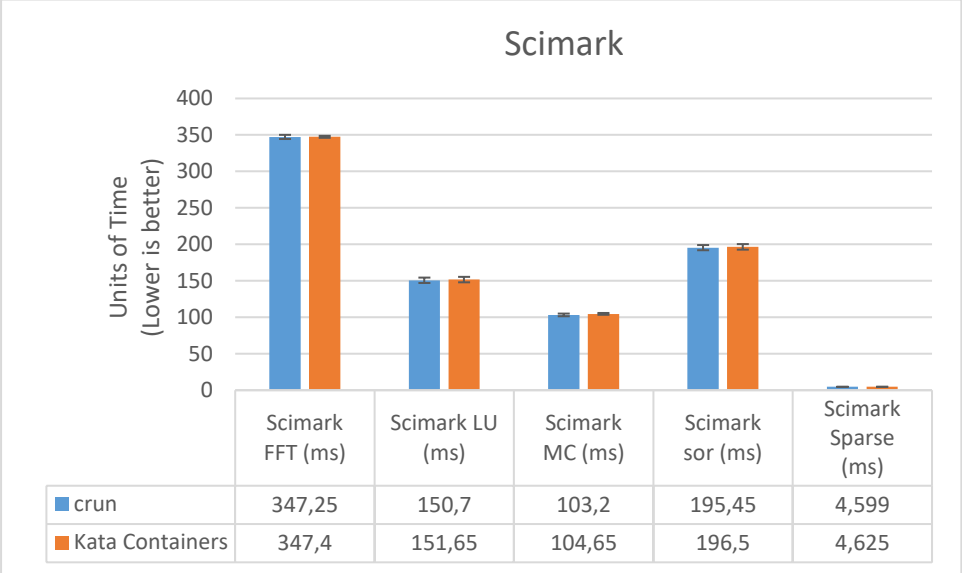
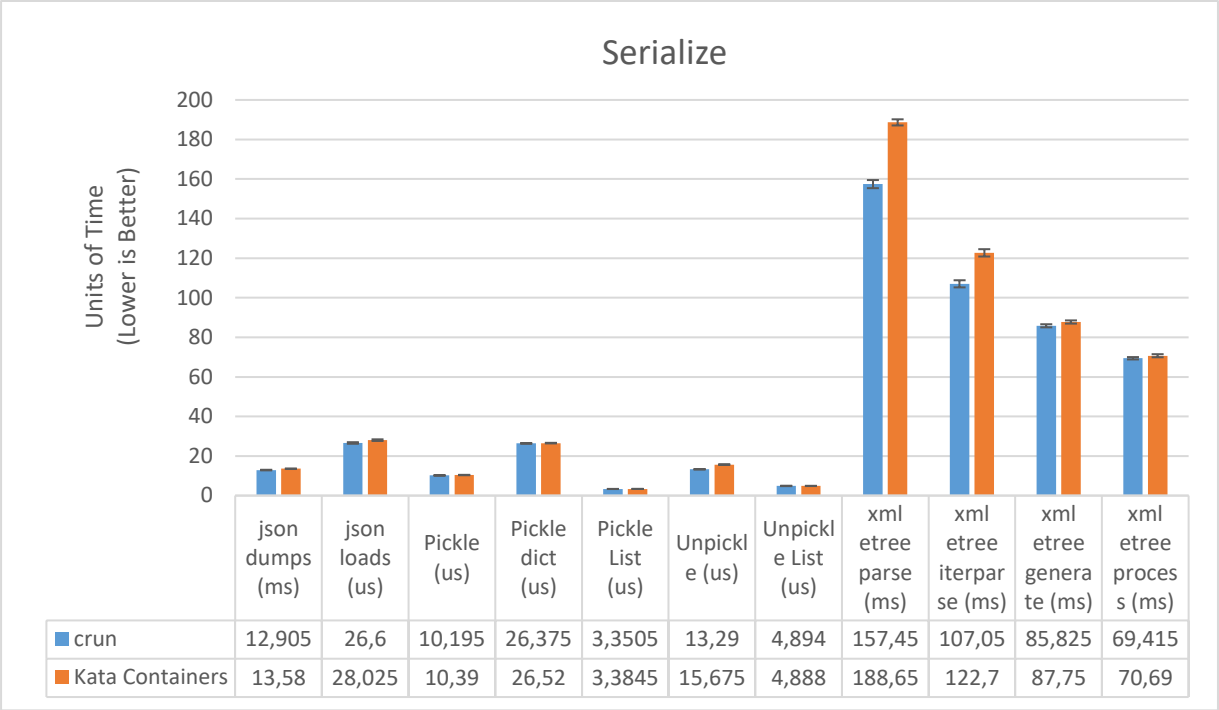


Figure 4 - SciMark performance

Barely any overhead can be noted in these computational benchmarks. The biggest difference in SciMark is Monte Carlo (MC), with a difference of *1.41%* while the other have *<1%*. Total for this category is *0.64%*.

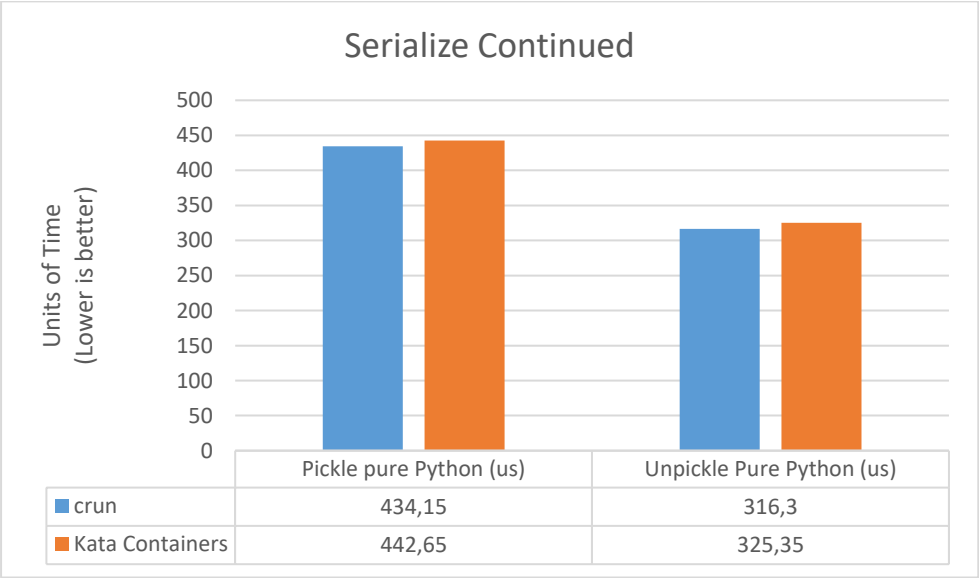
7.5 Serialize



	json dumps (ms)	json loads (us)	Pickle (us)	Pickle dict (us)	Pickle List (us)	Unpickle (us)	Unpickle List (us)	xml etree parse (ms)	xml etree iterparse (ms)	xml etree generate (ms)	xml etree process (ms)
crun	12,905	26,6	10,195	26,375	3,3505	13,29	4,894	157,45	107,05	85,825	69,415
Kata Containers	13,58	28,025	10,39	26,52	3,3845	15,675	4,888	188,65	122,7	87,75	70,69

Figure 5 - Serialize performance (note the different unit of time in parenthesis)

A small overhead can be seen in the JSON tests. The benchmark shows a difference of *5.23%* and *5.36%* in the dumps and loads respectively. The biggest overhead here is in the XML parsing, with *19.82%* in XML parse and *14.62%* in iterparse. All of the pickle modules have a non significant overhead, with the exception of Unpickle, where the overhead is just under *18%*. While some show a quite high performance regression, the total in this category in figure 5, is *6.18%*.



	Pickle pure Python (us)	Unpickle Pure Python (us)
crun	434,15	316,3
Kata Containers	442,65	325,35

Figure 6 - Serializing continued – these were put in a separate diagram for readability.

As seen in figure 6, the two benchmarks use pure python to obtain their goals. Using the unpickle pure Python-module instead of the cPickle - which is used in the test in figure 5 - to unpickle, only produce a *2.86%* overhead in contrast to *17.95%*.

7.6 SQL

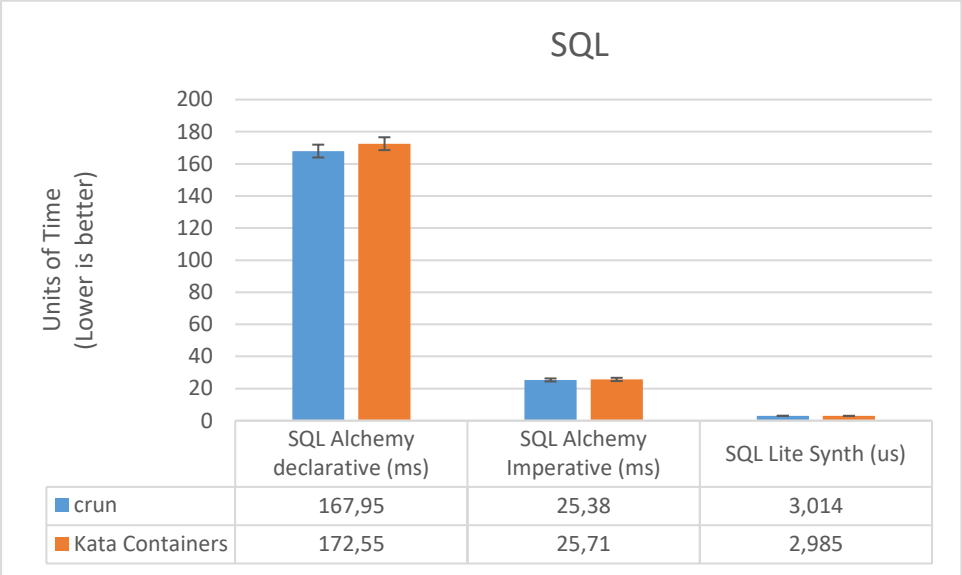


Figure 7 - SQL performance (note the different unit of time in parenthesis)

This category test different modules of SQL available for Python. Again, the difference here is pretty non-significant, showing that Kata-containers can perform in some cases on par with crun. Lite Synth even have a better performance (*-0.96%*), though calculating in the standard deviation the result is similar. Total for the SQL module is *1.01%* performance overhead.

7.7 SymPy

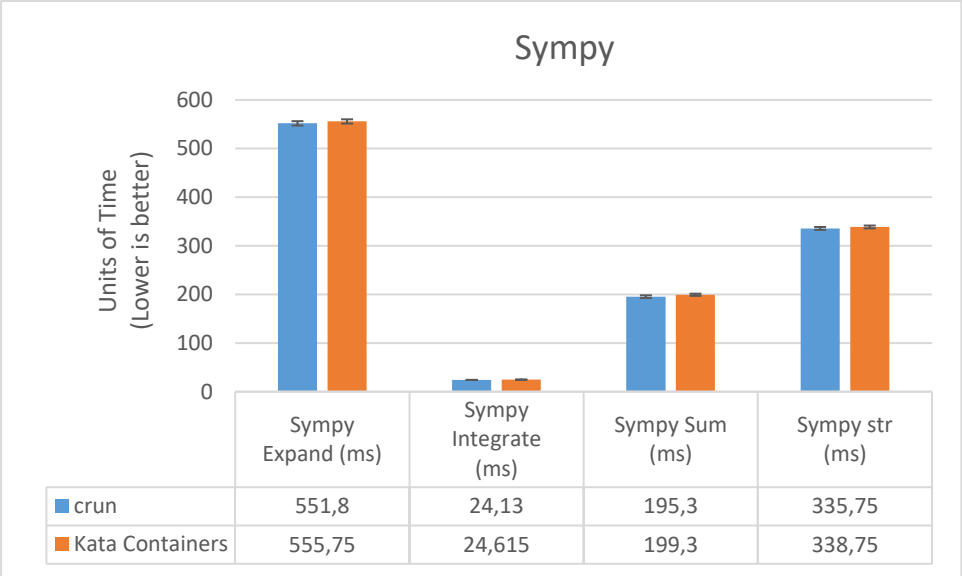


Figure 8 - SymPy performance

SymPy is a library for symbolic mathematics. These see more overhead than previous computing libraries. SymPy Integrate and Sum overhead is just over *2%* while Expand and String have less than *1%* overhead. Total overhead for this is *1.41%*.

7.8 Regex

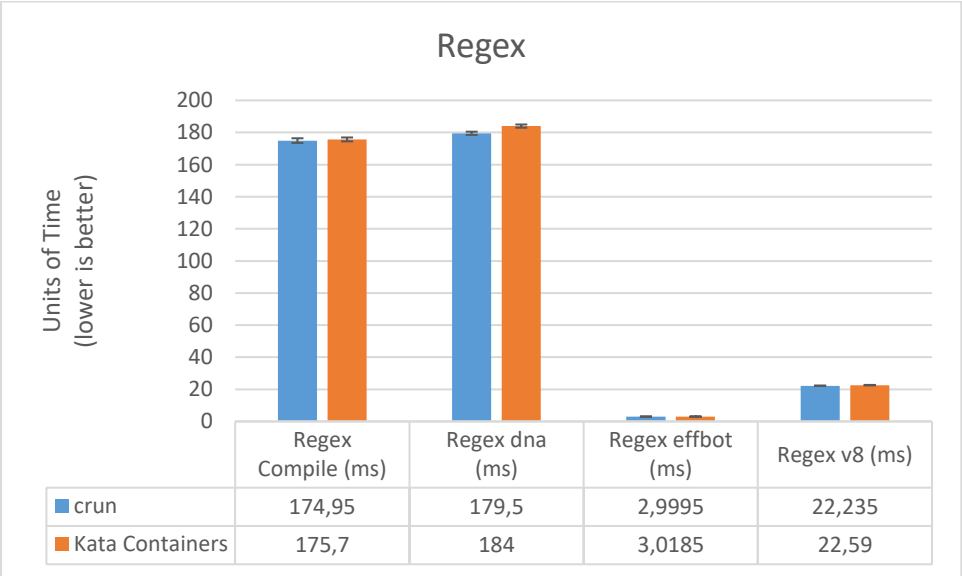
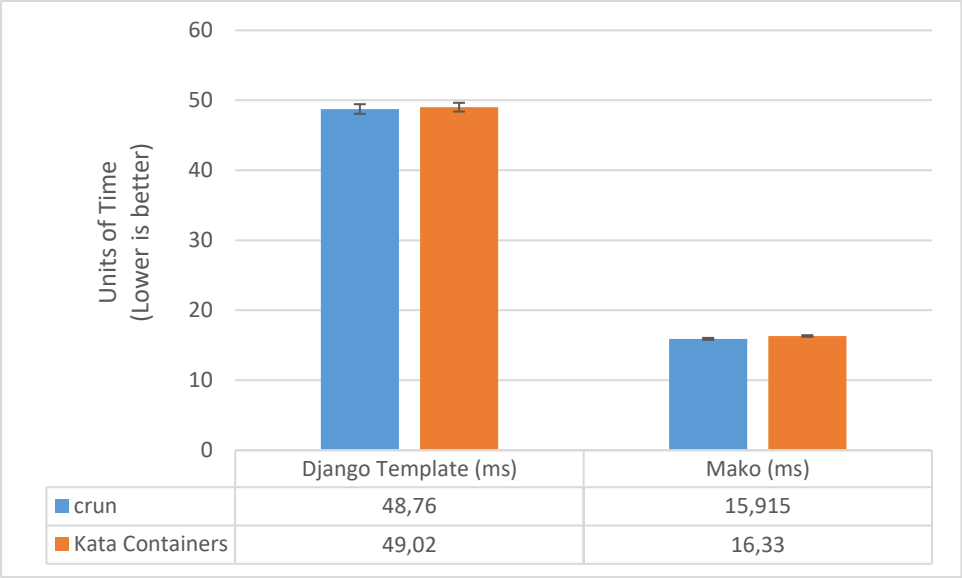


Figure 9 - Regex performance

The overhead of Python regex is negligible in effbot and v8, but a slightly bigger overhead is spotted in the DNA module, where the overhead is *2.51%*. Still the total overhead is only at *1.29%*.

7.9 Template

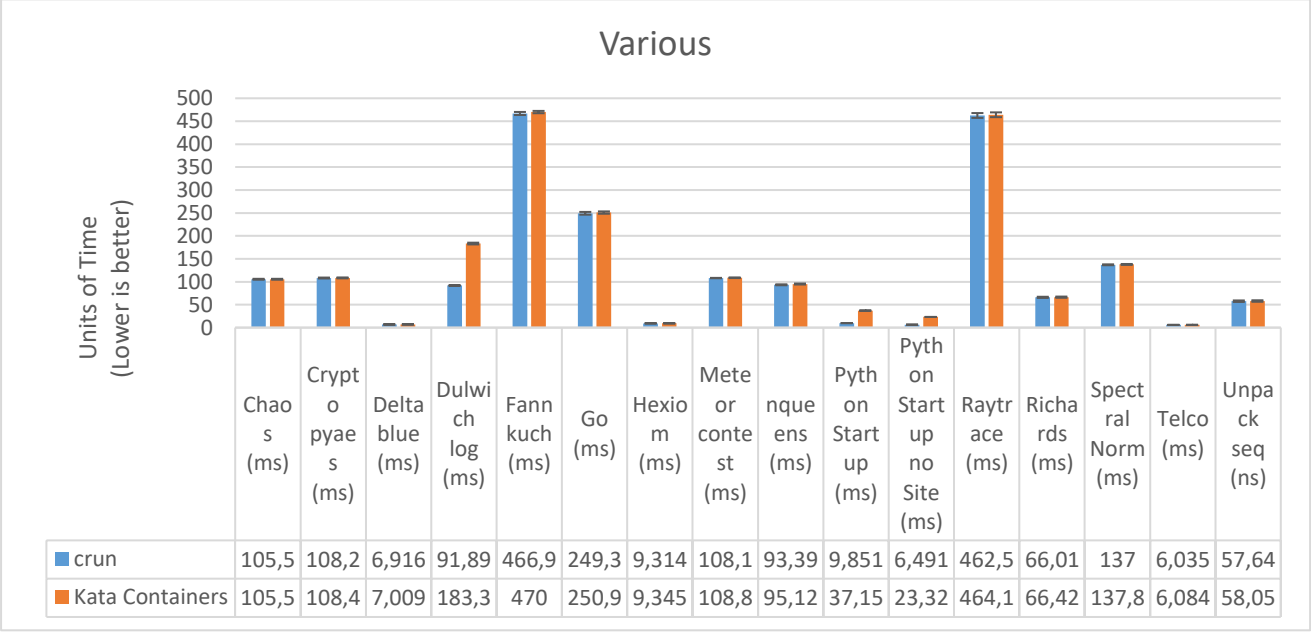


	Django Template (ms)	Mako (ms)
crun	48,76	15,915
Kata Containers	49,02	16,33

Figure 10 - Performance of templating libraries

Templating is often used in website generation. These two tests have the same type of goal: to generate a 150x150-cell HTML table. Both Django and Mako is used for templating in various website around the globe. In these test Django have a *<1%* and Mako *2.61%*, generating a total overhead of *1.57%*.

7.10 Various



	Chaos (ms)	Cryptopyaes (ms)	Deltablue (ms)	Dulwich log (ms)	Fannkuch (ms)	Go (ms)	Hexiom (ms)	Meteor contest (ms)	nqueens (ms)	Python Startup (ms)	Python Startup no Site (ms)	Raytrace (ms)	Richards (ms)	Spectral Norm (ms)	Telco (ms)	Unpack seq (ns)
crun	105,5	108,2	6,916	91,89	466,9	249,3	9,314	108,1	93,39	9,851	6,491	462,5	66,01	137	6,035	57,64
Kata Containers	105,5	108,4	7,009	183,3	470	250,9	9,345	108,8	95,12	37,15	23,32	464,1	66,42	137,8	6,084	58,05

Figure 11 - Performance of various tests

These are the tests that could not fit in to a category. All tests except 3 have an overhead of *~1%*. These 3 tests are Python start-up with and without the site module, and Dulwich Log. The different tests of Python start-up measure the time to load Python Path and libraries. Dulwich Log is a way to access GitHub, without using git, instead the module uses Python access GitHub.

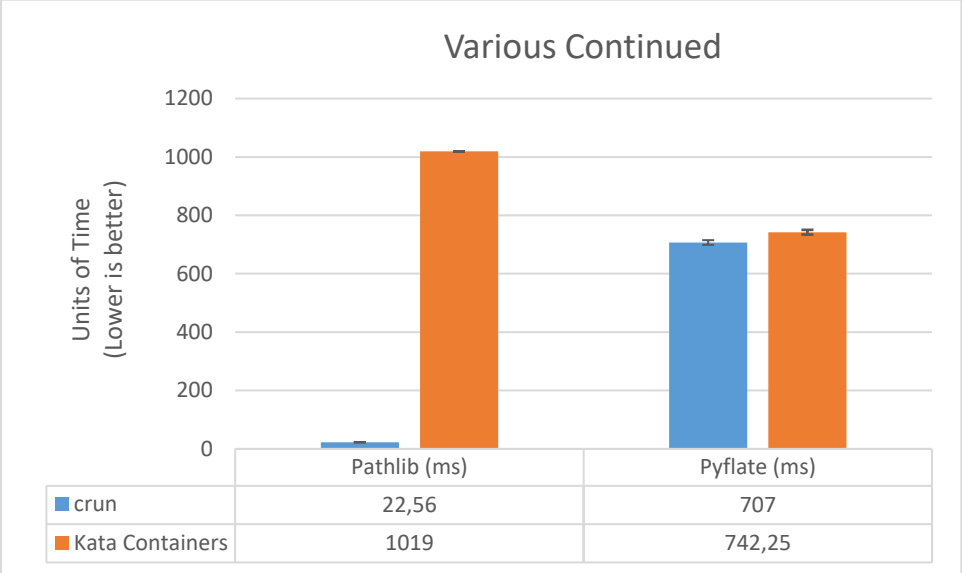


Figure 12 - Performance of pathlib and Pyflate

The pathlib module is ~44x slower in Kata containers. Analysing the source code for the pathlib module, there are three functions. The first generates filenames, the second creates temporary folder that hold files and the third is the benchmarking where the code executes loops that iterates over the created files and folders, with the modules glob() and iterdir() from the pathlib library. Glob() collects the matching pathname of a pattern and iterdir() gives the path objects of a selected directory. This means accessing the filesystem in a Kata container comes with a heavy overhead.

7.11 Test Summary

Name of test (Unit of Time)	Crun mean	Crun Std Dev	Kata mean	Kata Std Dev	Difference
2to3 (ms)	345,95	1,95	548,55	2,2	58,56%
Chameleon (ms)	9,073	0,128	9,1865	0,1275	1,25%
Chaos (ms)	105,45	1	105,45	1	0,00%
Crypto pyaes (ms)	108,2	1	108,4	1	0,18%
Deltablue (ms)	6,916	0,146	7,009	0,1735	1,34%
Django template (ms)	48,76	0,685	49,02	0,625	0,53%
Dulwich log (ms)	91,885	0,855	183,3	1,9	99,49%
Fannkuch (ms)	466,85	3,35	469,95	2,55	0,66%
Float (ms)	117,95	1,1	118,9	1,1	0,81%
Go (ms)	249,25	3	250,85	2,55	0,64%
Hexiom (ms)	9,3135	0,085	9,3445	0,0865	0,33%
JSON dumps (ms)	12,905	0,15	13,58	0,135	5,23%
JSON loads (us)	26,6	0,395	28,025	0,41	5,36%
Logging format (us)	9,38	0,172	9,4065	0,1715	0,28%
Logging silent (ns)	176,231	5,507	188,45	5,35	6,93%
Logging simple (us)	8,543	0,148	8,353	0,146	-2,22%
Mako (ms)	15,915	0,14	16,33	0,11	2,61%
Meteor contest (ms)	108,05	0,4	108,75	0,7	0,65%
Nbody (ms)	125,65	1,1	125,7	1,1	0,04%
Nqueens (ms)	93,39	0,995	95,115	1,1	1,85%
Pathlib (ms)	22,56	0,26	1019	0	4416,84%
Pickle (us)	10,195	0,115	10,39	0,1	1,91%
Pickle dict (us)	26,375	0,12	26,52	0,15	0,55%
Pickle list (us)	3,3505	0,0315	3,3845	0,034	1,01%
Pickle pure Python (us)	434,15	5,95	442,65	6,1	1,96%
Pidigits (ms)	180,05	0,2	181,1	0,5	0,58%
Pyflate (ms)	707	8,1	742,25	8,05	4,99%
Python startup (ms)	9,851	0,0345	37,15	0,235	277,12%
Python startup no site (ms)	6,4905	0,0205	23,32	0,1	259,29%
Raytrace (ms)	462,5	5,25	464,05	5,2	0,34%
Regex compile (ms)	174,95	1,45	175,7	1,2	0,43%
Regex DNA (ms)	179,5	1	184	1	2,51%
Regex effbot (ms)	2,9995	0,028	3,0185	0,022	0,63%
Regex v8 (ms)	22,235	0,1655	22,59	0,16	1,60%
Richards (ms)	66,005	1,275	66,42	1,32	0,63%
Scimark fft (ms)	347,25	2,85	347,4	1,35	0,04%
Scimark lu (ms)	150,7	3,7	151,65	3,75	0,63%
Scimark mc (ms)	103,2	1,85	104,65	1,25	1,41%
Scimark sor (ms)	195,45	3,55	196,5	3,8	0,54%
Scimark sparse mat mult (ms)	4,599	0,0165	4,625	0,0305	0,57%
Spectral norm (ms)	137	0,95	137,75	1	0,55%
SQL alch declarative (ms)	167,95	4	172,55	4	2,74%
SQL alch imperative (ms)	25,38	0,94	25,71	1,005	1,30%
Sqllite synth (us)	3,014	0,066	2,985	0,063	-0,96%
Sympy expand (ms)	551,8	4,5	555,75	4,35	0,72%
Sympy integrate (ms)	24,13	0,2	24,615	0,22	2,01%
Sympy sum (ms)	195,3	2,8	199,3	2,2	2,05%

Name of Test (Unit of Time)	Crun Mean	Crun Std Dev	Kata Mean	Kata Std Dev	Difference
Sympy str (ms)	335,75	2,95	338,75	2,95	0,89%
Telco (ms)	6,035	0,12	6,084	0,112	0,81%
Tornado http (ms)	166,5	3,05	154,45	3,15	-7,24%
Unpack seq (ns)	57,635	1,635	58,045	1,59	0,71%
Unpickle (us)	13,29	0,15	15,675	0,115	17,95%
Unpickle list (us)	4,894	0,1005	4,888	0,0625	-0,12%
Unpickle pure Python (us)	316,3	3,7	325,35	4,8	2,86%
XML etree parse (ms)	157,45	2,05	188,65	1,55	19,82%
XML etree iterparse (ms)	107,05	1,8	122,7	1,85	14,62%
XML etree generate (ms)	85,825	0,79	87,75	0,81	2,24%
XML etree process (ms)	69,415	0,6	70,69	0,78	1,84%
TOTAL	**54,59**		**63,33**		**16,00%**

8 Conclusions

This study set out to answer to research questions: (1) *"How does python code differ in terms of execution time in secure runtimes opposed to a default runtime?"* and (2) *"What modules and libraries differ?"*.

The results show that in most scenarios, there is no significant overhead (<5%) between native and virtualized runtimes. With that stated, the total overhead of all tests compared is *16%*. That answers the first research question. With that said, there are some cases where the overhead is significant enough to raise the question if an application could satisfy end-users in a more secure runtime.

The second research question is answered with, while all modules vary, the variation have a wide range. Depending on the job and type of job, the overhead can be from insignificant to large. Test that is heavily affected by a light weight VM, is pathlib, Serializing, Python start-up.

This research verifies previous research, that CPU heavy tasks, such as mathematical computation, i.e., SciMark and Math suites, show a little overhead. In addition, this study shows some larger bottlenecks, such as, pathlib, Python start-up times, and some modules in serializing (XML and JSON). An application that uses a combination of these, will see a larger overhead. This can explain why previous applications tested show a performance regression with around *40%* in Kata Containers.

One thing to note about some of the tests are that they have a performance increase in Kata Containers. Why this is can't be answered at the moment and further testing should be done. Two educated guesses could be that: (a) The performance issues could be related to crun. (b)

Minì and Giorgio (2020) research had some cases where Kata had a slight advantage in case of performance. This should of course be researched more to come to an empiric conclusion.

The extra layer of protection in Kata Containers, makes it a safer alternative no doubt, and doesn't have to take a toll in every case. The toll is most noticeable in cases that access the filesystem, the pathlib module, and Python start up times. And even in some cases it can outperform crun. The findings in this study could help when developing and deploy Python in Kata Containers. The areas that were tested are extensive and gives some direction what modules can be deployed in Kata Containers without a large overhead. For example: Web services that use templating tools Django or Mako will barely notice any difference if deployed in Kata Containers. Same with scientific computing that uses libraries tested in this study.

9 Future work

Same set of tests can be run on other secure runtimes, OCI-compatible or not, to spot which modules have a lower performance regression. There could also be benchmarks made on different compiled versions of both kernel and Python, to spot if any treatments can be done on the modules with a large overhead.

More research could be done with other experiment setups, one possible way could run the Python (or any other programming language or implementation of Python) code with profilers to measure where bottlenecks happen in running code. And from there work out treatments to decrease overhead between Kata and crun.

References

Amela, R., Ramon-Cortes, C., Ejarque, J., Conejero, J., & Badia, R. (2017). Enabling Python to execute efficiently in heterogeneous distributed infrastructures with PyCOMPSs. *Proceedings Of The 7Th Workshop On Python For High-Performance And Scientific Computing*. doi: 10.1145/3149869.3149870

Cabodi, G., Giorgio, D. A., & Minì, G. (2020). *OS-level virtualization with Linux containers: process isolation mechanisms and performance analysis of last generation container runtime* (Doctoral dissertation, Politecnico di Torino).

Cao, L. (2017). Data Science. *ACM Computing Surveys*, *50*(3), 1-42. doi: 10.1145/3076253

Casalicchio, E., & Iannucci, S. (2020). The state-of-the-art in container technologies: Application, orchestration and security. *Concurrency And Computation: Practice And Experience*, *32*(17). doi: 10.1002/cpe.5668

capabilities(7) - Linux manual page. (2021). Retrieved 15 August 2021, from https://man7.org/linux/man-pages/man7/capabilities.7.html

Changes/PythonNoSemanticInterpositionSpeedup - Fedora Project Wiki. (2021). Retrieved 15 August 2021, from https://fedoraproject.org/wiki/Changes/PythonNoSemanticInterpositionSpeedup

GitHub - containers/crun: A fast and lightweight fully featured OCI runtime and C library for running containers. (2021). Retrieved 2 September 2021, from https://github.com/containers/crun

Fleming, P., & Wallace, J. (1986). How not to lie with statistics: the correct way to summarize benchmark results. *Communications Of The ACM*, *29*(3), 218-221. doi: 10.1145/5666.5673

Rebuilding Python for improved performance on IBM Power Systems. (2021). Retrieved 15 August 2021, from https://developer.ibm.com/technologies/linux/tutorials/rebuild-python-for-improved-perf-on-power/)

Li, Z., Kihl, M., Lu, Q., & Andersson, J. (2017). Performance Overhead Comparison between Hypervisor and Container Based Virtualization. *2017 IEEE 31St International Conference On Advanced Information Networking And Applications (AINA)*. doi: 10.1109/aina.2017.79

Kata Containers - Open Source Container Runtime Software. (2021). Retrieved 15 August 2021, from https://katacontainers.io/

namespaces(7) - Linux manual page. (2021). Retrieved 15 August 2021, from https://man7.org/linux/man-pages/man7/namespaces.7.html

McGee, J. (2021). The 6 steps of the container lifecycle - Cloud computing news. Retrieved 2 September 2021, from https://www.ibm.com/blogs/cloud-computing/2016/02/08/the-6-steps-of-the-container-lifecycle/

Mehringer, S., & Barker, B. (2020). Using Containers to Create More Interactive Online Training and Education Materials. *Practice And Experience In Advanced Research Computing*. doi: 10.1145/3311790.3396641

Open Container Initiative - Open Container Initiative. (2021). Retrieved 15 August 2021, from https://opencontainers.org

Stack Overflow Developer Survey 2021. (2021). Retrieved 2 September 2021, from https://insights.stackoverflow.com/survey/2021

Welcome to Python.org. (2021). Retrieved 3 September 2021, from https://www.python.org/doc/

The Python Performance Benchmark Suite — Python Performance Benchmark Suite 1.0.3 documentation. (2021). Retrieved 15 August 2021, from https://pyperformance.readthedocs.io/index.html

Varela, J., Wehn, N., Desmettre, S., & Korn, R. (2017). Real-Time Financial Risk Measurement of Dynamic Complex Portfolios with Python and PyOpenCL. *Proceedings Of The 7Th Workshop On Python For High-Performance And Scientific Computing*. doi: 10.1145/3149869.3149872

Viktorsson, W., Klein, C., & Tordsson, J. (2020, November). Security-Performance Trade-offs of Kubernetes Container Runtimes. In *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)* (pp. 1-4). IEEE.

What is a Container? | Docker. (2021). Retrieved 15 August 2021, from https://www.docker.com/resources/what-container

Yoshimura, T., Nakazawa, R., & Chiba, T. (2020). ImageJockey: A Framework for Container Performance Engineering. *2020 IEEE 13Th International Conference On Cloud Computing (CLOUD)*. doi: 10.1109/cloud49709.2020.00043

Wohlin, C., Runeson, P., Host, M., Ohlsson, M., Regnell, B., & Wesslen, A. (2012). *Experimentation in Software Engineering*.

Appendix A

```
FROM ubuntu:20.04 AS builder-image

ARG DEBIAN_FRONTEND=noninteractive

RUN apt-get update && apt-get install --no-install-
recommends -y \ python3.9 \
python3.9-dev \
python3.9-venv \
python3-pip \
python3-wheel \
build-essential \
&&   apt-get clean \
&& rm -rf /var/lib/apt/lists/*

RUN python3.9 -m venv /home/b18frebj/venv
ENV PATH="/home/b18frebj/venv/bin$PATH"

# Build Layer
FROM ubuntu:20.04 AS runner-image

RUN apt-get update && apt-get install --no-install-
recommends -y \
python3.9 \
python3.9-venv \
&& apt-get clean \
&& rm -rf /var/lib/apt/lists/*

RUN useradd --create-home b18frebj
COPY --from=builder-image /home/b18frebj/venv
/home/b18frebj/venv

ENV PYTHONUNBUFFERED=1
ENV VIRTUAL_ENV=/home/b18frebj/venv

RUN python3 -m pip3 install pyperformance
```