Native WebLab

Safe Execution of Native Code in WebLab

B. Crielaard
C. Bruin
T. Aerts

Technische Universiteit Delft

TU Delft

Native WebLab

Safe Execution of Native Code in WebLab

by

B. Crielaard
C. Bruin
T. Aerts

to obtain the degree of Bachelor of Science
at the Delft University of Technology

Project duration:	May 1, 2017 – July 4, 2017	
Thesis committee:	Prof.dr. E. Visser,	TU Delft, coach
	Ir. D. M. Groenewegen,	TU Delft, Programming Languages Group
	Ir. O. W. Visser,	TU Delft, Bachelor Project Coordinator

TU Delft

Preface

B. Crielaard
C. Bruin
T. Aerts
Delft, June 2017

This thesis is the conclusion of the bachelor project for the Delft University of Technology. This report and the code itself were written by Taico Aerts, Chiel Bruin and Bram Crielaard. Over the course of three months, we have designed and built a system for safely natively executing code on WebLab, with a focus on Python. WebLab is a platform built and maintained at the TU Delft for executing student code online in a safe environment, while also allowing for automatic grading.

We would like to thank Danny Groenewegen for the support during the project, and the often instant responses to questions. We would also like to thank Eelco Visser for giving us the opportunity to work on a project we are all extremely interested in.

We would also like to thank Micha de Rijk, for writing a Python script that demonstrated a way to circumvent one of our security checks.

Finally, we would like to thank Otto Visser for his support during the project. When a situation arose where the continuation of our project was at risk, he immediately helped resolve the problems.

Summary

WebLab is a service used and provided by the Delft University of Technology. It is used by the university to automatically and safely execute and test student code. The current system makes use of the Java Virtual Machine (JVM). This means that in its current implementation the system only supports languages which can be executed on the JVM. However, with the growing popularity of Python as a scientific language, a need was found for WebLab to run languages such as Python. The JVM port of Python was deemed inadequate, as it does not support essential libraries. As a result, a redesign of the current system was requested to add full support for Python while allowing for other languages to easily be supported in the future.

We researched how a new system could be designed and implemented, which guarantees the safety and scalability required for WebLab. During the research phase, it was determined Docker provides all the security features we were planning on implementing ourselves. However, for added security the Docker containers are run on a Linux server, whose kernel has been hardened using grsecurity.

The decision to use Docker also had the added benefit of the system being easily scalable. More servers running Docker can be added to the system by simply adding their information to the settings. After this, these servers are used to run user code (also referred to as tasks). As a result, the main bottleneck of the system is the CPU of the server running the system itself.

Because the user code could overwrite the tests designed by the course instructor, a process was designed to guarantee users can not tamper with their test results. Altered test methods which should fail are added, the order of all methods is shuffled, and the names are obfuscated. Because of this, a guarantee can be made that tampering can be recognised if a user were to return answers at variance.

The system was thoroughly tested using unit tests to test code itself, and scripts which flood it with new tasks to test the reliability. It was found that the system was able to correctly handle peaks of up to four hundred tasks submitted at the exact same time, which was deemed sufficient by the client.

Contents

Glossary

WebLab frontend

The WebLab website on which users can submit tasks.

API

Application Programming Interface, an entry point for a piece of (library)code by which other systems can interact with it.

Container

"A Docker container is an efficient, lightweight, self-contained system which guarantees that software will always run the same, regardless of where it's deployed" [11].

Docker

A software platform which makes use of containers.

Docker daemon

The docker daemon process is running on the host and manages images and containers. It is also called Docker Engine.

Docker-machine

A tool to manage Docker instances running on multiple machines.

Image

"A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings." [12].

JVM

The Java Virtual Machine. The JVM is the environment in which Java programs run.

Machine

A machine is an entity capable of executing tasks. Each machine has a set amount of task slots equal to the number of tasks it can execute concurrently. Please note that a machine does not need to be a physical entity. A machine might well be a virtual machine.

Task

A task consists of either (1) running the user code and sending back all output, (2) running the user-defined tests and sending back all output and the test score, or (3) running the specification tests and sending back the test score. A task has a limited amount of resources it can use and has a limited execution time (timeout).

1 Introduction

WebLab is a system where students can write and test code using an online interface. Course coordinators can create assignments and exams. They define tests which make sure the student code meets requirements.

In its current implementation, WebLab does not support programming languages which do not run in the Java Virtual Machine (JVM). As certain courses are currently requesting support for these types of languages, a solution had to be found to make this possible. WebLab currently offers support for non-JVM languages by using a JVM based implementation of these languages. However, these implementations often lack support for some of the language's features which makes them unsuitable for WebLab.

At the time of writing this report, no systems exist which fulfil all requirements for WebLab, such as security and scalability. There are systems which allow users to run arbitrary code, but these either suffer from extensive limitations or from security issues. For example, the user can not see the result of `print` statements in their code, making debugging very difficult, or the user can bring down an entire backend server by running a fork bomb. These issues make them unsuitable for WebLab.

The purpose of this project is to design a system which enables WebLab to support arbitrary languages, while still being able to guarantee security and reliability. To limit the scope of this project, the focus was laid on the programming language Python. Support for Python has been requested by numerous course instructors as Python is a programming language which is widely used amongst different disciplines.

The aim of this report is to inform the reader on the design of our system and the design decisions made based on research done in the research phase. There are some extra features and improvements that are currently not implemented. For these features, we included a set of recommendations on how to implement them, with warnings about possible security flaws where necessary.

2

Problem Definition and Analysis

Before we could start working on the implementation, we first had to define the requirements for the project. The requirements make the aim of the project clear and allow the scope to be determined. The main requirements were based on the level of security, scalability and flexibility that is needed for the system. An overview of the full requirements, as created at the start of the project, is given in Appendix B. This chapter analyses these core requirements in section 2.2 to 2.4. These sections are based on the findings made during the research phase of the project that are described in more detail in the research report included as Appendix C. Lastly, the choice of a programming language for the project, based on the requirements, is explained in section 2.5.

2.1. The Problem

For its assignments, WebLab currently only supports languages that run on the JVM. As there is a need to support more languages, some workarounds were made to support Python and C. These solutions were however based on converting the source into a language that can be run on the JVM. This was impractical as it resulted in weird error messages or rendered frequently used libraries inoperable. Therefore a need was created for a system that allowed to run the code of non-JVM languages natively. One of the languages currently not fully supported is Python. As Python is a programming language that is widely used at the TU Delft, there is a high demand for Python courses on WebLab. It is for that reason that the main focus of our project was to add support for Python. If possible, we should build the system in such a way that it provides the groundwork for the support of more languages in the future.

2.2. Security

The main focus of the project is to be able to execute Python code in a safe way. This way a user with malicious intent cannot breach the server, cause harm to other users, or improve their scores. Ensuring a level of security that is as high as possible required us to research security features of Python and of the system itself. Based on the research, we concluded that Python is inherently unsafe and that it must be protected at kernel level by some form of process containment.

2.3. Scalability

Another major requirement was that the system must be scalable to support the high peak loads. For example, during exams, these loads can be up to several hundred users at a time. Therefore we decided on an implementation that can be deployed over multiple machines. This way, additional machines can be added when more users use the system. Docker, specifically docker-machine, was found to be a good candidate, as it also offers process containment. Therefore, it has all the features we need for both security and scalability.

2.4. Flexibility

Preferably the system created for running the Python processes should be flexible enough to support more programming languages. This way WebLab could be easily expanded to support other programming languages that might be wanted in the future. This was not a 'must have' requirement, but when a solution

was found that could facilitate this, that solution should be preferred. By using Docker containers for the scalability and security, it is also possible to run different images on those containers. This allows for a high flexibility in a way that we could support any possible programming language by creating a Docker image for said language.

2.5. Programming Languages

The choice of a programming language depends on many factors. In our case, an important requirement was that we needed to be able to interact with Docker and Docker-machine. The most important factor that determined our choice of language was our client. As the current system is written mostly in Scala and Java, one of these languages must be used to make sure the systems stays maintainable. From these two, the client preferred Java. Therefore this language was chosen as the language that is used for the development of our system. This language also has some advantages for us. Most notably that we are all familiar with it, and we also knew all the tools that we can use for this language that help in maintaining high code quality. Finally, there are also multiple APIs available to interact with Docker from Java.

3

Design - Task Scheduling

This chapter explains the design of the task scheduling component of our system and elaborates on the design decisions for the different components of the task scheduling. This chapter covers the entire lifecycle from receiving a new task to having handled said task.

3.1. Definitions

Acceptor

A thread which takes tasks from the new task queue and sends them to a machine.

Destruction queue

A queue containing tasks that are waiting to be destroyed.

Destructor

A thread which takes tasks from a destruction queue. After taking a completed task, the destructor ensures that the correct information is returned to the user and that any resources used by that task are disposed of.

Event handler

A thread which listens for events from Docker. Whenever an event is received that a task has completed execution, the event handler adds that task to a destruction queue.

Folder watcher

A thread which listens for events sent by the file system. A folder watcher waits for modifications to files or for the creation of new files. The folder watcher processes these events and either creates a new task or updates an existing task with new information.

New task queue

The queue containing all tasks that are waiting to be executed.

System load

"A measure of the amount of computational work that a computer system performs" [43].

Timeout checker

A thread which checks if there are tasks that have timed out.

Watchdog

A thread which monitors the state of the entire system. The watchdog redistributes resources amongst components where needed.

3.2. System Overview

This section provides a few overview images of our system. Figure 3.1 shows the interaction between the WebLab frontend and our system. Figure 3.2 shows all the different components of a machine and the task flow between them. Figure 3.3 shows how a single successful task goes through the different components of our system.

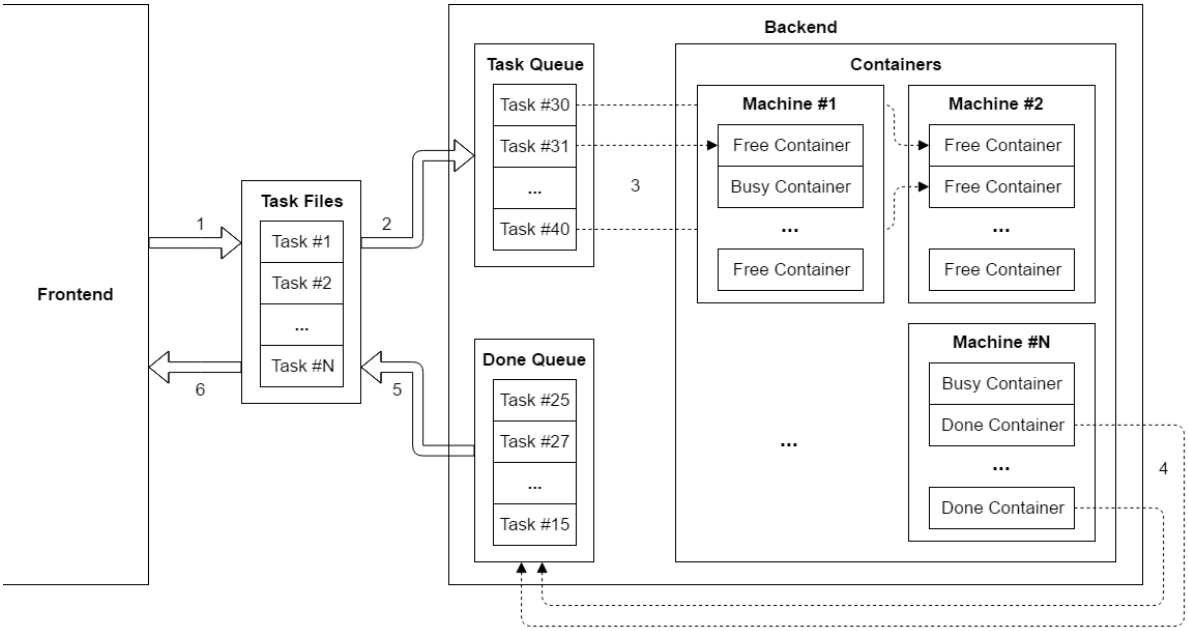


Figure 3.1: How tasks go from the frontend to the backend and back.
1) New task files are created
2) Tasks queued (Folder watcher - section 3.3)
3) Tasks scheduled in containers (Acceptors - section 3.4)
4) Done containers to done queue (Event Handler - section 3.5)
5) Results written to task files (Destructors - section 3.7)
6) Frontend reads results

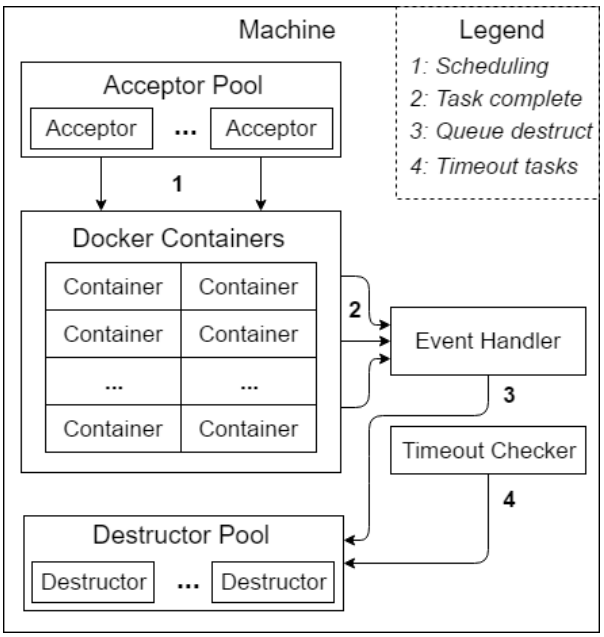


Figure 3.2: Overview of all the components used for a single machine.

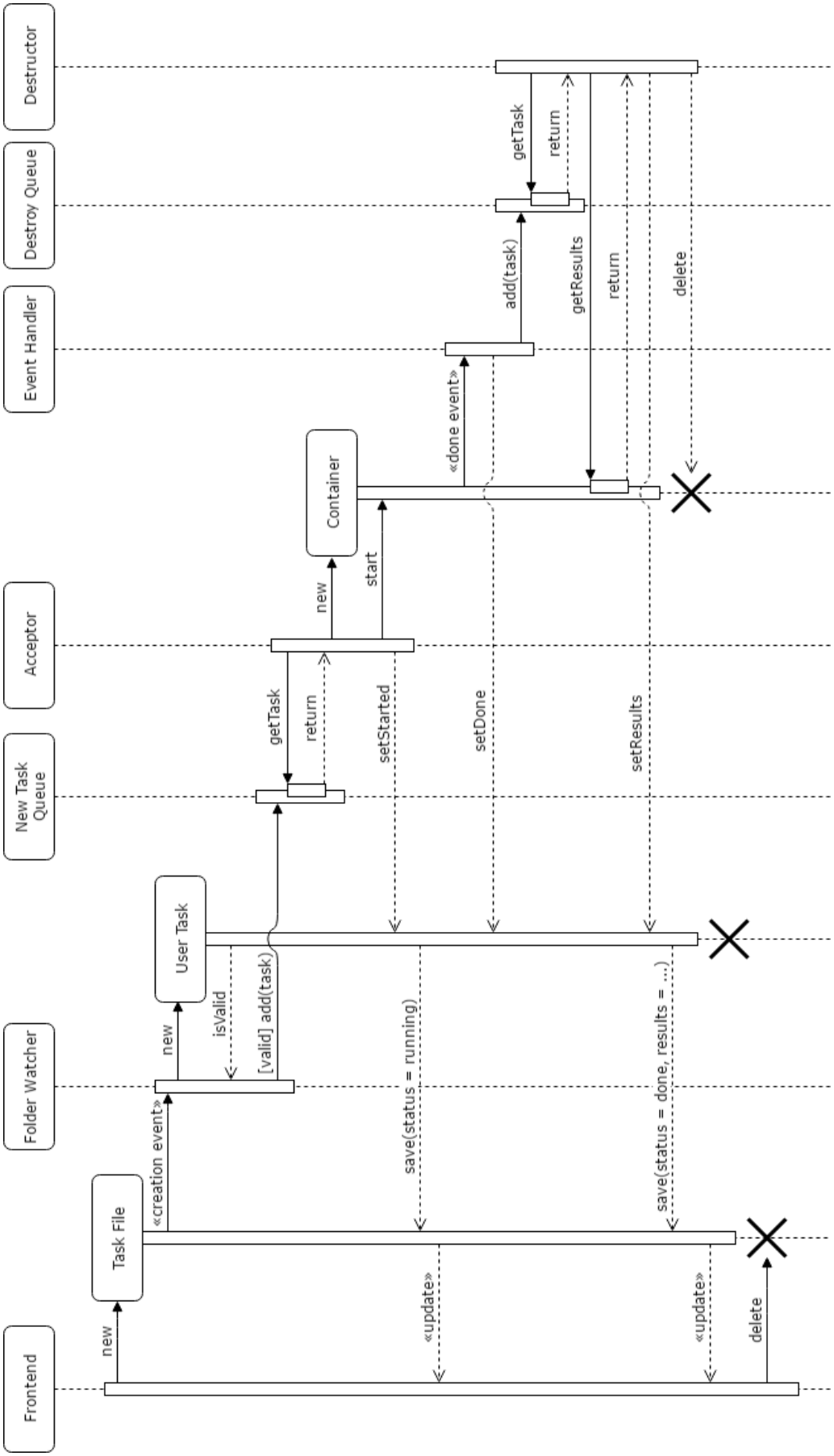


Figure 3.3: Sequence diagram for handling a single successful task.

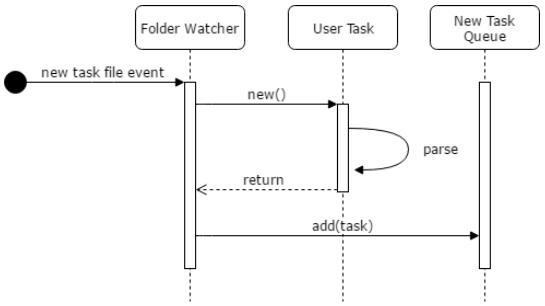
3.3. Folder Watcher

The first component of our system is the folder watcher. The WebLab frontend places task files in a specific directory. The folder watcher listens for file events in that directory by using the WatchService API in Java. The Java WatchService API, in turn, hooks into the Linux inotify functionality to listen for the file events.

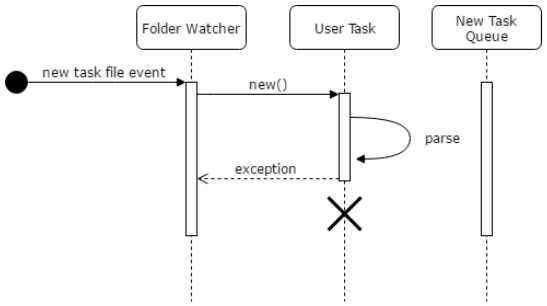
We use a single folder watcher which listens for file creation events. Because of the way task modifications are implemented in the frontend, we do not need to listen for any other events (see subsection 3.3.2). The folder watcher underwent a few changes over the course of the project, which are explained in subsection 3.10.1.

3.3.1. New Tasks

Whenever a task file is created by the WebLab frontend, a file creation event is issued which the folder watcher handles. We first check if the corresponding task is already in the system. If not, the folder watcher will load the task by parsing the XML and will then add the task to the new task queue. Under normal circumstances, the frontend should always supply us with correct task files. In the event that the task file is invalid for some reason, we reject the task and log a warning message. By correctly handling invalid tasks, we reduce the chance of potential problems.



Valid task



Invalid task

3.3.2. Updating Tasks

Whenever a task is cancelled in the WebLab frontend, the corresponding task file is updated with the `cancelled` status. A modification to a task file is implemented in the frontend by replacing the original file with the updated file. We receive this as a file creation event. If we find that a task is already in our system, we know that it must have been a modification.

Just as for new tasks, we first parse the XML file. If the status is anything but the `cancelled` status, we ignore the change. The frontend only changes the status of a task to `cancelled`, so any other statuses must have been written by ourselves. If the status has indeed been set to `cancelled` then there are two possible scenarios. If the task was not yet scheduled and is still in the new task queue, it will be disposed of by an acceptor. Otherwise, if the task is already running on a container, then it is added to the destruction queue. A destructor will then kill the task and remove the container. Figure 3.4 shows how tasks are cancelled. For simplicity sake, the event is displayed as a modification event.

3.4. Accepting Tasks

Every machine has a pool of so-called acceptors. An acceptor is a thread that actively takes tasks from the new task queue and starts each task in a container on the machine. The pool will always contain at least one acceptor and at most one acceptor for every task slot.

An acceptor lies dormant whenever there are no new tasks, consuming only a small amount of system memory and no other system resources. Whenever a new task is added to the new task queue, one acceptor is selected at random and is woken up to handle the new task.

The number of acceptors in the pool is increased and decreased dynamically based on the number of tasks in the new task queue. If the queue has a lot of tasks waiting, more acceptors are added. If the queue is empty, an acceptor is removed from the pool periodically. The thresholds for these dynamic adjustments are configurable. The dynamic scaling is performed by the watchdog (see section 3.8).

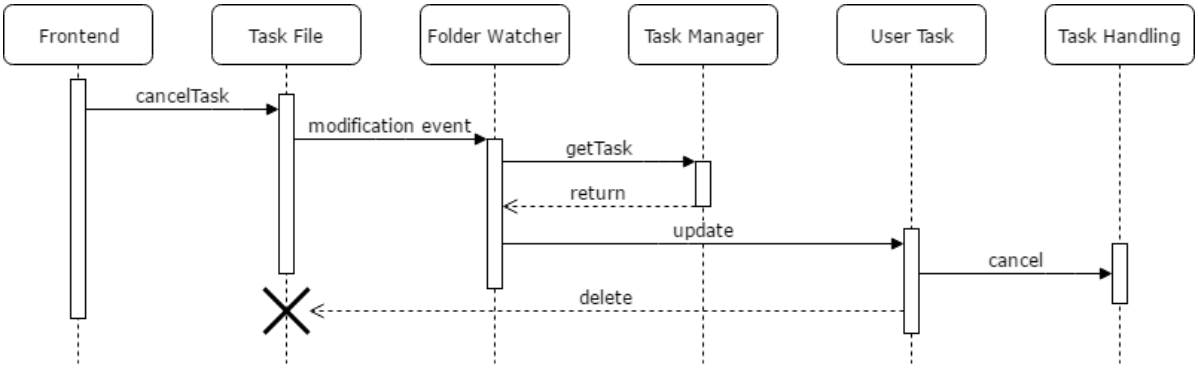


Figure 3.4: Sequence diagram of task cancellation

3.4.1. Equal Tasks

The aim is that every task gets the same amount of resources, regardless of the number of users. This limiting of resources is enforced on the containers. Regardless of the load of the system, a task will never get more CPU time or memory than the set limit. This makes the output of the system more deterministic. If a task takes 10 seconds to complete under low system load, it should also take 10 seconds to complete under high system load. By employing this mechanism, we avoid the scenario where a solution would be marked as passing all tests at one moment, while it would be marked as failing all tests at another moment.

3.5. Event Handler

Every machine has an event handler. This event handler listens for the so-called “container die” event from a machine. When it receives such an event, it schedules the associated task for destruction by placing it in the destruction queue.

The “container die” event is sent in two different situations. The first situation is if the program running inside of the container stopped running. This would mean the program either completed normally or that it encountered an error and stopped abnormally. The second situation is if the container was killed. Whenever a destructor removes the container of a machine, it also kills the container if it is still alive. This happens whenever the timeout of a task has passed. Because the task was already handled by a different component whenever such a situation occurs, we simply ignore the event.

3.6. Timeout Checker

Every machine also has a timeout checker. Its job is to ensure that all tasks finish eventually. When the timeout of a task has passed, the timeout checker schedules it for destruction by placing it in the destruction queue. A destructor will then inform the user that a timeout has occurred and will kill the container running the task. Figure 3.5 shows a sequence diagram of a task timing out.

It is possible for a task to complete execution after being flagged as timed out, but before the container of the task is removed. In this situation, we will still dispose of the task and any results, even if we could give the actual output to the user. This further increases consistency between runs.

3.7. Destructing Tasks

Just as every machine has a pool of acceptors, every machine also has a pool of destructors. Destructors are responsible for finishing a task. They will ensure that the correct output is sent to the user and that the container in which the task was run is removed. If the task is still running when the container is being removed, it is automatically killed.

The destructor pool has a destruction queue to which tasks can be added to be destructed. The queue follows the FIFO principle, that is, the first task that enters the queue is the first task that leaves the queue.

The watchdog increases and decreases the number of destructors in the destructor pool when needed, just like it does for the acceptor pool. The number of destructors is determined based on how long it takes for a task to go from done to the point where it is accepted by a destructor.

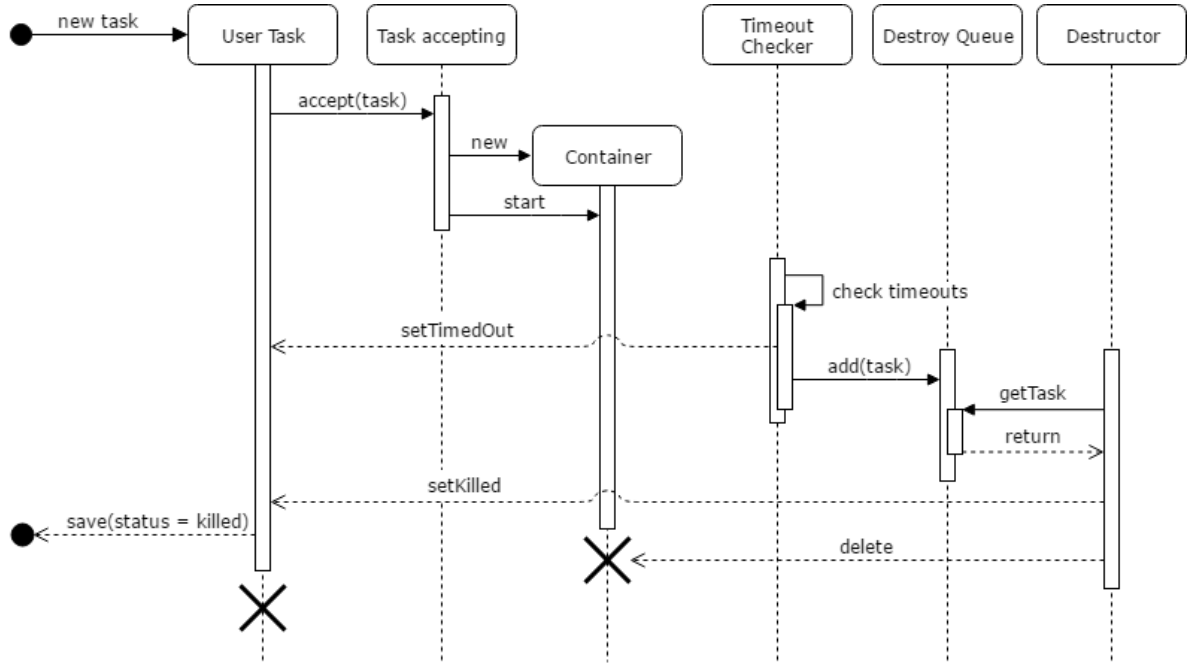


Figure 3.5: Sequence diagram for a single task that gets timed out.

3.8. Watchdog

The watchdog is responsible for monitoring the system, and for reallocating resources across the different system components and machines. This section explains how and when the watchdog scales different components as well as the design ideas behind that scaling.

3.8.1. Load Monitoring

Because we use a task queue, we can easily determine the size of the queue. If the task queue progressively grows in size, that means that more tasks are coming in than we can handle. We are also able to identify the bottleneck in the system. If there are still free slots available on our machines, we would need to create more acceptors. If tasks in a destruction queue need to wait for a long time, we need more destructors. Almost all parts of the system can be scaled up or down dynamically, to accommodate for any bottlenecks.

The only component that we cannot scale dynamically is the number of machines. The only way to create machines out of thin air would be by using a service like Amazon Elastic Compute Cloud. However, this is out of scope for this project.

Our system will show warnings to indicate that the number of machines is not sufficient, and could be easily extended to notify system administrators using other methods. As moments of high system load (like exams) can be known in advance, system administrators can make sure there are enough machines available beforehand.

3.8.2. Stale Container Checker

The watchdog is also responsible for ensuring that container cleanup happens eventually. Once in a while, the watchdog will do a check for stale containers. A container is considered stale whenever it was created so long ago that the timeout should have already passed at least four times. Any container that was not removed after its timeout has passed multiple times must have been left by us by mistake. This situation could occur if the connection to the machine is lost for a small period of time. By running this periodic stale container cleanup, we ensure that no garbage is accumulated on machines.

3.8.3. Scaling Components

The watchdog scales the number of acceptors and destructors per machine in order. This is best explained with an example. The same mechanism that is used for acceptors also applies to destructors.

Let's assume the scenario where there are two machines in the system, machine A and machine B. Each

machine has one acceptor. Whenever the watchdog decides to scale up the number of acceptors, it will start by increasing the number of acceptors of machine A. Only when machine A reaches its acceptor limit, will the watchdog start to increase the number of acceptors of machine B.

Similarly, whenever the watchdog decides to scale down the number of acceptors, it will start by decreasing the number of acceptors of machine A. Only when the number of acceptors for machine A has reached the minimum, will the watchdog start to decrease the number of acceptors of machine B.

3.8.4. Design Decision

The main idea behind this design is that machines can be added to the system whenever more capacity is needed and that machines can be removed whenever the capacity is not needed. We make full use of a machine before switching to the next machine.

Our system does not perform any kind of load balancing because it does not have to. Because we give every task the same amount of resources and because every machine has a set number of tasks, we never overload machines. Tasks will simply wait longer in the task queue.

The benefit is that, again, our system remains consistent under heavy load. Instead of trying to run many more tasks and failing all or most tasks (due to timeouts), our system will keep running tasks at a fixed rate. Tasks that are too old will eventually be cancelled by the WebLab frontend or by the user.

3.9. The Lifecycle of a Task

The examples below show how a single task can propagate through our system.
The lifecycle of a successful task (see Figure 3.3):

1. A user submits a new task.
2. The WebLab frontend adds a new task file in the task folder.
3. A folder watcher receives an event for the new task and adds it to the new task queue.
4. An acceptor accepts the task and schedules it on a machine in a free container.
5. The task is executed.
6. An event handler receives an event that the task has completed execution and adds the task to a destruction queue.
7. A destructor takes the task from the destruction queue, retrieves the output of the task, sends the correct information to the user and releases all resources used by the task.

The lifecycle of an unsuccessful task (see Figure 3.5):

1. A user submits a new task.
2. The WebLab frontend adds a new task file in the task folder.
3. A folder watcher receives an event for the new task and adds it to the new task queue.
4. An acceptor accepts the task and schedules it on a machine in a free container.
5. The task is executed.
6. After a configurable amount of seconds, the timeout checker marks the task as timed out and adds the task to a destruction queue.
7. A destructor kills the running task, sends to the user that the task has timed out and deletes any leftover files.

3.10. Design Changes

Along the way, our design changed as we ran into problems or thought of different solutions. The largest design changes are explained in this section.

3.10.1. Split Folder Watcher

Initially, we had a single folder watcher which listened for both creation and modification events. The first tests showed that this approach worked fine. However, when we started testing with a large number of tasks at once we suddenly received the `overflow` event. Whenever the inotify event buffer is full and a new event is placed in it, it clears the event buffer and adds the `overflow` event. For us, this means that we will miss events for the creation of tasks and as an effect never run the tasks.

To debug this problem, we registered an inotify listener from the command line. We submitted a task from the WebLab frontend and looked at the events that were reported. We found that there were a lot of additional

events that were created because of the way the WebLab frontend creates the task files, but also the way we modified task files ourselves. We made some modifications to limit the number of events and decided to split the folder watcher into two folder watchers either listening for creation or for modification events.

The folder watcher listening for modification events still gets most of the events and can get an overflow. However, missing modification events means missing that tasks are being cancelled in the worst case. Further testing also showed that we are capable of handling the events of 400 tasks per second without problems, which is a sensible limit. With loads of more than 400 tasks per second, the WebLab frontend should perform load balancing across different instances of our backend. Preferably, additional backends should be run on different servers.

Later on, we found that the frontend modifies task files by replacing them with new files. As we receive file replacement as a file creation event, the folder watcher listening for modification was not actually needed. Luckily, the changes required were minor. We only needed to modify the code handling new tasks to call the code for modification events, whenever the task was already in the system.

3.10.2. Acceptor Design Change

Initially, we designed the system with unequal workers in mind. In this situation, every machine can define what tasks it can accept. The idea behind this is as follows. If there is an exam where all tasks will be Python, it would make sense to add a machine that can only execute Python tasks.

The logic we implemented for this is to create a queue where acceptors poll items from the queue. If an acceptor polls an item that it cannot execute, it notifies another acceptor and waits until there is a new item at the head of the queue. The second acceptor will then wake up and do the same until we reach an acceptor that is capable of accepting the task.

We created a mechanism for detecting if there is no machine that can accept the task. After retrieving a task, we evaluate if the task is still "valid". We implemented valid as: not cancelled and of a type that at least one machine can execute. If the task is invalid we remove the task from the queue.

Unfortunately, this approach does have a large problem. Assume that we have 100 acceptors waiting for tasks. 99 acceptors can accept tasks of type A and only one acceptor can accept tasks of type B. When an item of type B enters the queue, an acceptor is chosen at random and checks if it can accept the task. If not, it notifies another random thread and goes to waiting on a different condition. In the worst case, we would sequentially wake up 99 acceptors that cannot accept the task before the single acceptor that can accept the task is woken up.

There is also a second problem. As soon as the task is removed, *ALL* acceptors wake up sequentially to switch their wait condition from "wait for different item" to "wait for new item". Now let us go back to our exam scenario. Imagine that during the exam, a student that does not participate in the exam submits a task for a different course, in the programming language C. As we have set up more machines for Python, the above scenario suddenly becomes very real. In effect, this single student can decrease the performance of the entire system significantly.

When there are multiple exams of different types and we have specialised machines, the odds are better, but still not good. We still have the problem that we need to wake up lots of acceptors that cannot accept the task, simply to have them switch their blocking condition.

In the end, we decided that it would make much more sense to make every machine equal. If every machine can accept every task, our logic would be much simpler and quicker. Adding more machines will never decrease the performance of our system. The only downside is that every machine needs to have all images available in the system. As Docker already uses layer sharing to limit the amount of disk space and memory required for images that only differ slightly, there should be little to no additional strain on the executing machines.

4

Design - Docker

Based on our findings presented in the research report, we decided on a system using Docker containers. The research report can be found in Appendix C. This chapter will give an overview of the Docker portion of our system design. It also includes additional research that was needed for the correct limiting of system resources, which is discussed in section 4.3. In section 4.4 we discuss what images are and how we use them. In section 4.5 we discuss how we can make the docker images as small as possible. We also discuss the APIs we used to communicate with Docker from within Java in section 4.1 and section 4.2.

4.1. Docker Engine API

We had to make a choice about how we would interface with Docker. Docker lists two libraries for Java in its documentation: docker-client[1] and docker-java[2]. Docker-client is maintained and used by Spotify and offers an object based API to Docker. Docker-java is maintained by KostyaSha and offers a slightly more command-based API. Both APIs would suit our needs and are actively maintained.

We ultimately decided to go with the Spotify docker-client. The fact that a large company like Spotify uses the API means that there will be little to no bugs that impact the usability in large-scale environments. This is also what we see if we look at the issues on GitHub. Docker-client has mostly minor issues that would not affect our usage, while docker-java seems to have some issues that could affect us. [3]

4.2. Docker-machine API

We also needed to interface with docker-machine. Docker-machine doesn't provide any API at all. The only way to interface with docker-machine is by providing commands to its command-line interface. As such, we have created an API that sends docker-machine commands to bash. We have implemented this into a single class called `DockerMachineAPI` that we can use to access the functionalities of docker-machine.

4.3. Resource Limiting Limitations

As already discussed thoroughly in subsection C.1.1 in our research report, we decided on using the built-in Docker container features to limit the resources a program has access to. However, the resource limiting functions provided by Docker do not fully cover our requirements.

For memory management the functions were adequate. We were able to correctly limit swap and memory usage. By the setting `memory-swap=-1` , we disabled swap memory. There is a soft and a hard memory limit. Under normal circumstances, processes can use memory up to their hard limit. Whenever the Docker daemon needs more memory, the soft limit is enforced. The soft limit is created using the `memory-reservation` setting and `memory` created the hard limit. The exact value of this limit is calculated based on the total memory available on the machine it is running on and the maximum number of tasks it can process concurrently.

Managing the CPU usage of a process was less straightforward. The period in `cpu-period` turned out to be the maximum length of a continuous execution period and not the total execution time [27]. This way

[1] `https://github.com/spotify/docker-client`
[2] `https://github.com/docker-java/docker-java`
[3] For example: `https://github.com/docker-java/docker-java/issues/770`

it can only be used in combination with `cpu-quota` to limit the maximum CPU usage. This feature is very useful for giving the n users on a server at most $100/n$ percent of the available CPU. In combination with setting `cpu-shares`, we were able to create a fair scheduling for each process in terms of their maximum CPU usage. Therefore, we were able to prevent a single process from choking the server.

However, a piece of code can be run indefinitely within this system. A solution for this is the usage of the `timeout` command [3]. But as we use an API for starting the containers, this method cannot be applied in our case, because we do not run the `docker run` command ourselves. As a result, we have decided on using a separate thread that kills any tasks after a set amount of time called the timeout checker (see section 3.6).

With these limits in place, we have conducted multiple test runs to verify that the resources are correctly limited. These tests are described in section 7.3.

4.4. Images and Languages

Docker uses images as the building blocks of a container. The image specifies everything that should be available in the container. This includes software, e.g. Python, but also additional files that should be available to all containers for that image, e.g. our custom WebLabTestRunner for Python. A suitable Docker image with a test runner would be enough to add backend support for a particular programming language, but to avoid test tampering, a test parser and fuzzer would also need to be created.

There can be multiple different images for one programming language. For example, there can be one image which includes Python and module X, and another image which includes Python and module Y. By providing this functionality in the backend, limitations can be imposed on what is and isn't accessible for a particular course.

As we have focused on Python, we only provide one image by default. This is an image for Python 3 with often used modules like numpy and matplotlib included. Initially, we also wanted to add automated Docker image building for Python. We created scripts to build a Python image from a list of modules that should be available. Unfortunately, we were not able to add this functionality because of time constraints. Additional images, therefore, need to be created by a WebLab administrator. This process is explained in the manual in section F.1. In section 10.5 the option of implementing the feature of automatic image building and deployment is discussed.

4.5. Minimising Docker Images

For the system to be scalable, the images used by Docker need to be as small as possible. Initially, our Python image was approximately 790 MB in size. Therefore, we needed to research how to reduce the size of images to more reasonable sizes.

4.5.1. Layers

Before we can explain how the size of images can be reduced, we first need to explain how Docker images work. Docker images consist of multiple layers. Each of these layers can be shared between multiple images. In each layer, a part of the system is contained. For example, one layer can contain Python itself, while a different layer can contain the numpy module.

4.5.2. Squash

The most obvious way of trying to reduce the image size is by using something which is built into Docker itself. For this Docker includes the `-squash` parameter when building images. This parameter squashes all layers of the image into a single layer, as a result clearing cached files in layers. This, unfortunately, did not reduce the total file size of our images.

Another unfortunate result of using squash would be that layers can no longer be shared between images. Layer sharing would result in multiple images having the same base layers, without having to duplicate them. This would result in an overall smaller file size over multiple images, which is useful when different courses want to use different but similar images.

Because of these two reasons we decided not to use the `-squash` parameter to reduce the file size of our images.

4.5.3. Base Image

Another way to reduce the size of Docker images is by changing the base image. This entails selecting which version of Linux you want your container to use. We initially used the image made available by Python but deemed it was too large. After this, we tried to set one up using Alpine Linux, as it is a very minimalistic Linux installation. We were, unfortunately, unable to set up a working version of Python within this image.

After this, we attempted to use TinyCore Linux, as it is also extremely small, and provides an image which already has Python installed. This image ended up being 187MB, which was deemed reasonable for our project.

5

Design - Testing

The main feature of WebLab is the automated testing of user code. As the container given to each user by the backend includes the source files containing the test code, it becomes possible to inspect those tests. Therefore, there is a way to fake a perfect score without the correct solution. To address this problem, a system was needed to prevent people from faking their test results. Our system has been designed in such a way that it does not prevent people from doing so but does mark them as having attempted to cheat. This has been done in two parts. The first part, as explained in section 5.1, shows how we alter the tests written by course instructors to make it more difficult for students to recognise how their code is actually tested. The second part, as explained in section 5.2, shows how we generate test result files to send the results back to our system.

5.1. Altering Tests

To guarantee a fair examination of the user code, we determined it was necessary to alter the specification tests. Because students can theoretically make all tests pass by altering the specification tests, they should not be able to benefit from doing so.

5.1.1. Preventing Tampering

To achieve a state where we can safely say students can not cheat the tests, we designed a relatively simple system. This designed system shares similarities with both zero-knowledge proofs and a steganography technique called 'Chaffing and Winnowing'. A zero-knowledge proof is based on the idea of asking multiple questions to a black-box to verify that the box contains a valid solution for a problem without revealing the contents of the box [18]. Our system can be seen as a variant of this problem, as it is designed to be able to verify the given solution by running tests on it (asking questions) without exposing the test suite it is testing with. To achieve this we need to obfuscate the tests so the code that is tested cannot know the original test suite. A 'Chaffing and Winnowing'-like solution is used in our system to achieve this effect. In this steganography technique, extra data is added in such a way that a user can only distinguish the actual content from the noise when a key is known [41].

In our design, all test methods which contain an assert statement are duplicated. To create each additional test method, we take the complement of the last assert of the original test case. This means that a user could in no possible way, without altering the test suite or returning answers at variance, be able to pass both these tests. To make it so students can not just simply check the test names and find out which tests are altered, the order is shuffled and the names are anonymised. Now only the backend knows which tests are supposed to succeed or fail.

To implement this idea, we have designed the following system. As Python is currently the only implemented language we will explain the procedure for Python specification tests. The system is designed to easily be adaptable for new programming languages (see subsection F.1.4).

In Listing 1 we show the outline of what a specification test should look like. When a course instructor designs a specification test, they can write all their test methods between the comments at lines nine and nineteen. These comments indicate the start and end of the test method block. We use these comments to parse the

code. We have implemented a rather naive parser, for simplicity and speed. The code is read line by line. If a method definition is found, all following lines will be counted as belonging to that method. We take the complement of the last assert of each method (as seen in Table 5.1). Lastly, the order of the methods is shuffled and the names are anonymised. This means that even if people figure out the order in which to let tests fail and succeed, they would still fail on the next run as the order of the tests is not consistent.

The Python unit test framework has many asserts, most of which are almost never used [15]. Therefore, only the most used ones are flipped by the backend. This means only the following asserts create altered tests in our framework:

`assertEqual`	⟷	`assertNotEqual`
`assertTrue`	⟷	`assertFalse`
`assertIs`	⟷	`assertIsNot`
`assertIsNone`	⟷	`assertIsNotNone`
`assertIn`	⟷	`assertNotIn`
`assertIsInstance`	⟷	`assertNotIsInstance`

Table 5.1: Asserts and their complements

As a result, course instructors should include a test suite which contains a mix of these asserts, if they want their tests to be harder to tamper with.

```
1  import unittest
2  import weblabTestRunner
3  import solution
4
5  class TestSolution(unittest.TestCase):
6      def setUp(self):
7          # Do setup routine
8
9  # SPECTESTS START HERE
10     def testFirst(self):
11         # First test
12
13     def testSecond(self):
14         # Second test
15
16     def etc(self):
17         # More test methods are of course possible
18
19 # SPECTESTS END HERE
20
21     def tearDown(self:
22         # Do teardoum routine
23
24 if __name__ == '__main__':
25     unittest.main(testRunner=weblabTestRunner.TestRunner)
```

Listing 1: Sample code for specification tests

5.1.2. Why This Approach Works

This approach of altering tests works based on the principle of detecting a contradiction. Say we have a test *A*, as shown in Listing 2.

```
def testA(self):
    self.assertFalse(solution.getAnswer())
```

Listing 2: Test case *A*

After running *A* we have the results *B* and *C*. *B* represents the result of the normal run, *C* represents the result of the run where we have taken the complement, as seen in Listing 3.

```
def testNotA(self):
    self.assertTrue(solution.getAnswer())
```

Listing 3: The complement of test case *A*

The only influence the user has on these tests is the result for `solution.getAnswer()`. If the user were to answer inconsistently across runs, there are four possible scenarios. The only scenario where we know the user has cheated, is if he or she were to pass both of the tests. In a correct implementation, it would be impossible for something to be both true and false at the same time. However, it would be possible for the user to fail both tests (for example, if an exception is raised), or to pass only the complement of the original test (for example, if the question was implemented incorrectly).

This means we need to check the scenario where both results *B* and *C* are passing. This is as simple as checking for both *B* and *C* to be true. Please note that the tamper checking could fail if the specification tests do not properly reset the state after every test.

5.1.3. Guessing

With this system, it becomes impossible to fake the test results by saying that all tests have passed. The only way to find the correct solution which gives you a 100% score is to guess all the values correctly. In a normal case, without our tamper checking, the chance to correctly guess n yes/no-questions is equal to $P_n(n) = \frac{1}{2}^n$. For $n = 10$ this gives a chance of 0.098% of guessing the correct answers to all the tests. With the addition of our tamper check, the number of test cases doubles. Therefore the chance of guessing the correct solution with the tampering code enabled follows the following formula $P_t(n) = P_n(2n) = \frac{1}{2}^{2n}$. The chance of a correct guess for $n = 10$ now becomes 0.000095%.

When making random guesses, there is a 25% chance per test case to get flagged as tampering. From this we can calculate the expected number of times the tamper detection was triggered, for an original test suit size of n, to be $T(n) = \frac{1}{4} \times n$. So when applying guessing to determine the correct answers on a run where $n = 10$, it is expected that you are flagged 2.5 times. So running the multiple thousands of runs that are (expected to be) needed to get a 100% score when guessing, will most likely get you flagged.

In Python, it is possible to inspect and alter the code at runtime. Therefore, it is possible to create a script that finds the two test methods that are each other's counterparts. With such a script the tests can be run without triggering the tampering detection, by ensuring that each of the two linked tests cannot both return `true`. Now the tests can be run indefinitely without triggering the tampering detection. But due to the randomisation and the fact that it is impossible to know which test case should return what, it is still required to guess the correct values. As calculated above, the chances of guessing correctly are equal to P_n.

It must be noted that in the calculations made above, it is assumed that the ratio between positive and negative questions in the original test suite was 1:1. This results in a chance of $p = \frac{1}{2}$ to guess the correct answer. When this ratio shifts too much to one side, it becomes increasingly easier to guess the correct answers. With the knowledge of this preference for the correct answer, a guessing script can be created to be biased in the same way to increase its odds of guessing a question correctly. In an extreme case where $p = 1$ or $p = 0$, the guessing becomes trivial as it is known that all the positive/negative tests are expected to be the ones that de-

termine your result. It is therefore advised to create a spectest-suite in which the positive and negative tests are somewhat equally present.

5.1.4. Limitations

Because our parser is quite naive, there are some limitations for course instructors with regards to their tests.

- Python uses the same syntax for block comments as it does for multi-line strings. Because of this, we decided to drop support for multi-line strings in the test methods. This means instructors have to divide these strings into smaller single-line strings, followed by a new line character.
- Python supports decorators for test methods, which are similar to Java annotations. Because these appear before the method definition, they are not part of this method according to our definition. As a result, these are not supported. In our personal experience, we have not yet found a situation where we needed these decorators.

Naturally, both of these limitations can be solved. During our research, no Python parsing libraries for Java were found. Because of this, a full-fledged parser would have to be implemented. This would include writing a tokenizer for the language or porting an already existing one. It was deemed too time-consuming for us to implement a full parser.

5.2. Test Runners

By running the user code completely separate from the backend, there no longer is a direct way of gathering the test results from the run. There were three ways we could retrieve this data. The first and, in principle, most simple technique was to use the built-in functionalities of the docker-client API to receive the `stderr` and `stdout` from the containers. This raises the need to distinguish between the normal output and the output of the tests by our backend. This mixed data stream would, therefore, require additional tags to be printed to be able to make this distinction. This was a valid solution, but we opted against it as this would lead to a messy stream and, subsequently, messy code.

The second option was to send the results as a separate stream over some form of socket connection to the backend. This way we could receive three clean streams that could easily be parsed. This extra connection does add the need for some more complex code that opens and manages the connection. In addition to this, it would also be required to open more ports on the container, weakening its security.

We implemented the third option that was fully file-based. This means that the `stderr` and `stdout` are piped to two separate files and that the test results are also stored in a file. These three files can now be parsed easily to determine the results of the run and present this to the user. By using such an interface for retrieving test results, it also becomes easier to support new languages. Furthermore, this method has the advantage that it could support an unlimited number of other output formats (like images), as any new output format will just be an extra file that has to be retrieved from the container.

With this design structure in place, we only needed a way to actually write the test results to a file. To achieve this, we utilised a feature of the `unittest` framework for Python that allows us to add custom test runners. We constructed a runner which writes a file in a structured format, described in subsection F.1.5. [redacted]. This file is then parsed by the backend in order to present the results to the user.

6 Implementation

This chapter discusses how we integrated the new Docker-based backend into the WebLab systems. This integration is split up into two sections: the WebLab frontend (section 6.1) and the backend (section 6.2).

We also had to ensure that the current JVM-based backend(s) can run alongside our new Docker-based backend.

6.1. Frontend Integration

Besides creating an additional backend for WebLab, we also had to make changes to the WebLab frontend to support the new system. The WebLab frontend is written mainly in WebDSL, but some parts are written in Scala and Java. WebDSL is a *"Domain-Specific Language for Web Applications"* [20].

6.1.1. Adaptations for New Languages

First of all, we had to make some changes to add Python as a programming language for WebLab assignments. As there have already been experiments for using Jython to run Python in the JVM backends, we were able to reuse some code. We added templates for Python assignments according to the specifications of our Python parser for the test fuzzing (see subsection 5.1.1). The templates form a guide for the teacher creating the assignment by giving a short example of how the testing works. Listing 4 shows the specification test template.

```python
import unittest
from solution import Solution
import weblabTestRunner

class TestSolution(unittest.TestCase):
    def setUp(self):
        # ...

# Place all the tests between the START comment and the END comment.
# Do not remove the SPECTESTS comments

# SPECTESTS START HERE
    def test_name(self):
        # ...

    def test_othername(self):
        # ...

# SPECTESTS END HERE

if __name__ == "__main__":
    unittest.main(testRunner=weblabTestRunner.TestRunner)
```

Listing 4: Template for the Python specification tests

6.1.2. Tamper Flag

As explained in chapter 5, the Docker-based backend has the ability to detect test tampering. This tamper flagging has to be hidden from the student that performs the tampering but should be shown to all users that are allowed to grade the assignment. WebLab uses a status field to determine the result of a test run. As tampering behaves exactly like the `done` status in appearance, we decided that it would be best to make a new status called `tampered`.

Whenever a submission has a status of `tampered`, the frontend displays it exactly like a `done` status to the student, but it also sets the tampered field to true. WebDSL then handles this by saving this information to the database. Once the tampered flag has been obtained, there is no way for it to be removed again. The student will forever have the tampered flag for their submission of that particular assignment. As the tamper detection is still based on chance, flagging only the submission itself would not make sense. It does not matter how low the odds are. Given enough time, the student would be able to keep resubmitting until they get lucky enough to pass all tests without being flagged as tampered. By permanently keeping the tampered flag, we invite a grader to take a closer look at the student code. After all, the grader is the one who decides if the student code is a valid solution or not. See also our ethics discussion about this feature in section 9.3.

For graders, the tampered flag is shown in the submission overview as can be seen in Figure 6.1.

6.1.3. Course Settings

In addition to supporting multiple languages, the Docker-based backend supports multiple Docker images for each language (see section 4.4). For example, there can be a Docker image which has only the default Python installation and a separate Docker image with additional libraries installed like numpy and matplotlib. As such, each course should be able to specify the Docker image that should be used for the assignments. We added this as an additional tab to the course overview (see Figure 6.2). The page is only visible and accessible for the manager of the course.

As automated Docker image building was out of reach for the project, additional Docker images need to be created manually by a WebLab administrator. As such, using a different image for a course is something that needs to be done in collaboration with a WebLab administrator. This is the reason why we mark the image name as an advanced setting.

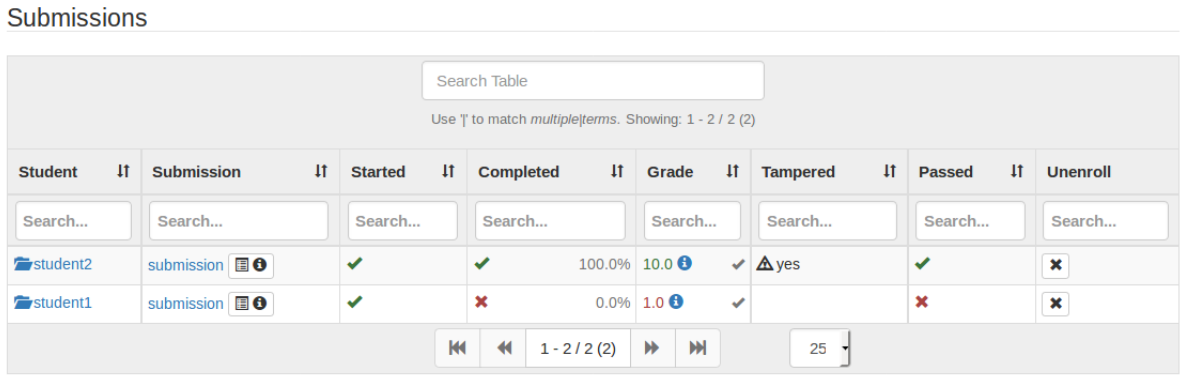


Figure 6.1: The tamper flag as it is displayed to assignment graders.

With the way the current backends parse the XML task files, changing the XML format (see Listing 5 for an example) would require modifications to all existing backends. To avoid this we decided to use the `code-name` field of the XML task file. The `code-name` field is only used for Scala assignments and is unused for all other languages. Whenever the course manager sets a different Docker image for the course, all tasks of that course will have that Docker image name set in the `code-name` field. Setting an invalid image means that tasks from the course will be rejected by the backend. If the field is empty, the default image for the language is used.

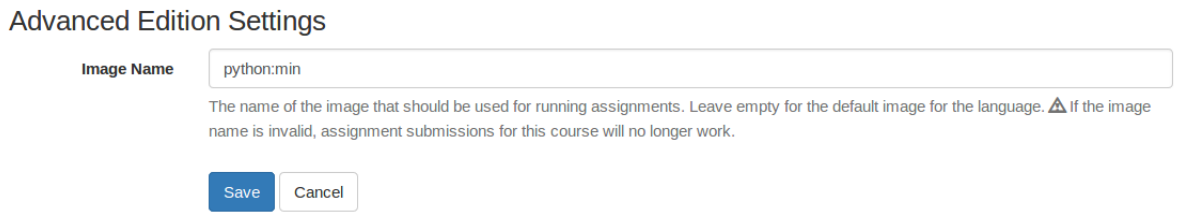


Figure 6.2: The advanced edition settings tab where the Docker image to be used by assignments can be set.

```
<task>
  <id>sometask</id>
  <data-dir>/tmp/webapp_docker/sometask</data-dir>
  <run></run>
  <code-name></code-name>
  <status>
    <pending></pending>
  </status>
  <compile-result></compile-result>
  <priv-compile-result></priv-compile-result>
  <exec-result>
    <num-cases>-1</num-cases>
    <num-failures>-1</num-failures>
  </exec-result>
  <recorded-out enabled="true"></recorded-out>
  <lang>Python</lang>
</task>
```

Listing 5: A WebLab XML task file

6.2. Backend Separation

To ensure that both the JVM-based backend and the new backend can run alongside each other, we decided to separate them completely. All tasks that should be run on the Docker-based backend are placed in a separate folder. This means that the Docker-based backend does not have to work in the same way as the current backends. It also means we did not have to make any changes to the current JVM-based backends.

6.2.1. Event-based vs Polling

The complete separation allows us to use the event-based folder watcher (see section 3.3) instead of the polling approach that the JVM-based backends use. The JVM-backends perform the following steps at a fixed interval:

1. Try to create lock file
   (a) If the lock file couldn't be created, wait 0.5 seconds and try again.
   (b) Otherwise, continue with step 2.
2. List all files in the task directory
3. Determine which files are new, which files have been removed and which files have been changed
4. Create new tasks, update existing tasks and dispose of removed tasks

This approach has a few disadvantages. The first disadvantage is that the backend has to lock the entire folder while it reads the task files. During this time only the backend that owns the lock file is allowed to read any files and no other backend can accept any new tasks.

The second disadvantage is that a backend accepting only specific languages still has to parse every task file to find out if it can or cannot execute the task. For example, the Java 8 backend still has to parse the XML files of tasks that require a Java 7 backend.

A third disadvantage is that the backends still have to check task files of tasks that are already being executed by a different backend. In order to mark a task file as "claimed", a backend has to change the `status` field in the XML from `pending` to `constructing`. Other backends then have to parse the file again to find that it has already been claimed and can be ignored by them.

All in all, this process introduces more wait time for tasks and a lot of unnecessary file reading and parsing. The event-based approach that we use allows us to react immediately to new tasks or changes. Whenever there are no tasks, we do not use any CPU time.

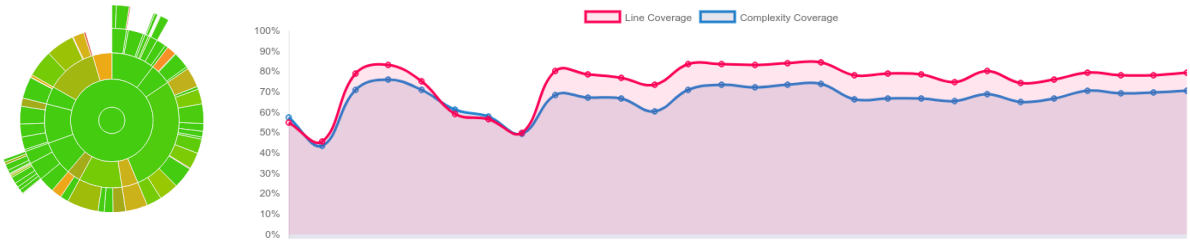
7

System Testing

In order to check if our system behaves correctly, we have conducted a number of tests. Most of these tests were automated and automatically run under continuous integration using Travis-ci. In addition to these automated (unit) tests, we have also conducted some manual tests to verify the workings of the system in simulated real world conditions. section 7.1 will discuss the coverage of the code we achieved by using unit testing. In section 7.2 and section 7.3, the manual tests are described and results are discussed.

7.1. Unit Testing

In order to automatically verify the workings of our system, we created many (unit) test cases. These tests are run automatically using the Travis CI[1] service for continuous integration. Besides testing the software components of our project, this also allows us to run regression tests after changes to the code, to ensure its behaviour did not change. However, for this tool to be useful, it is required to create a good set of test cases that cover all the system components. To measure the coverage we used two different tools, CodeCov[2] and Cobertura[3]. Cobertura rates our line coverage at 86% and the branch coverage at 75% and CodeCov sits slightly below that with scores of 80% and 72% on these metrics respectively.



(a) A 'sunburst' graph showing the coverage in the system components

(b) The history of our code coverage

Figure 7.1: Coverage graphs created by CodeCov

Figure 7.1a shows an overview of the coverage in all the different system components. Some classes show a lower coverage, which is caused by different factors. Most of these classes have interactions with resources outside of the system, like the `DockerMachineAPI` class which interfaces with docker-machine. These interactions are hard to simulate correctly with a test case, so we tested these interactions manually by verifying that, for example, machines were started at the correct moment. During the entire project, we aimed to add tests for each newly added piece of code. This can be seen in Figure 7.1b, where the coverage is plotted over the duration of the project. After the initial phase where the coverage fluctuated a bit, we succeeded in constantly keeping the coverage high. The first dip that can be seen was caused by adding a library without its

[1] `https://travis-ci.com/`
[2] `https://codecov.io`
[3] `https://cobertura.github.io/cobertura/`

tests. The second dip was caused by the addition of a major part of the task handling. This was a big task that took almost a full sprint. In the first part of the sprint, the focus was put on adding new functionalities rather than achieving an as high as possible test coverage. At this point, only some basic logic tests were added, which caused a slight dip. Near the end of the sprint, the focus shifted to the testing of the new components. This got the coverage back up to desired levels and made it possible to merge the new functionality into the `master` branch.

7.2. Reliability

Reliability is a very important metric when creating a tool on which numerous people need to rely. We need to know if the backend can handle sudden increases in load, e.g. during exams. As we can not test with an actual exam, three scripts were written which flood the system with tasks to simulate a heavy load. All of these scripts have a different purpose, which will be explained in the following subsections.

Flooder

The main script which was created simply floods the backend with one particular type of task. This is useful to simulate a worse case scenario load, for example, four hundred people submitting a fork bomb all at once. It was also used to check a best case scenario, for example when these four hundred people all submit a correct implementation. Having a single type of tasks makes it easier to determine run time, as there should be no variation between tasks.

Multiple File Flooder

Because a scenario in which everyone submits the same task is not realistic, a second script was used which floods multiple different types of tasks to the backend. This means we can simulate a more realistic load. Some tasks are actual implementations of the requested assignment and some contain only a single test case that passes immediately (no operations), while other tasks consist of fork bombs and infinite loops.

Canceller

Users can cancel tasks by resubmitting their assignment. If this happens, the backend should no longer run the task. The canceller randomly selects a given amount of tasks and cancels them, simulating users actively working on their assignment.

7.2.1. Results

After multiple days of testing, we came to the conclusion that the system is sufficient. The system was flooded multiple times with batches of four hundred tasks. These batches consisted of either infinite loops, fork bombs, correct assignments, no operations, or a combination of all four. In none of these runs did the backend crash. There were, however, certain irregularities during certain runs, that we were unable to reproduce. All tasks were handled in a timely and orderly fashion, indicating the system is reliable.

All four hundred of these tasks always completed in less than two minutes, even though we were running on a single computer with only two CPU cores. This means that during an exam more servers could be deployed to further increase the speed of the system.

7.3. Time Usage

To measure the amount of time the backend needs to handle the tasks, we used the scripts which were explained in section 7.2. We ran three different tests. All tests were run on a computer containing an `Intel i7 970` CPU clocked at `3.2GHz`, and a Crucial `M4-CT128M4SSD2` SSD.

The first run consisted of one hundred tasks containing no operations. This allowed us to measure how much time was spent on all components of the system besides the actual execution of the task. We used a single virtual machine for task execution, which was able to use two CPU cores and had twenty task slots available. The average time of this run can be seen in Figure 7.2.

The second run again consisted of one hundred tasks containing no operations. The hundred tasks were executed on two virtual machines, each of which was able to use one CPU core and had ten task slots available. The average time of the tasks in this run can be seen in Figure 7.3.

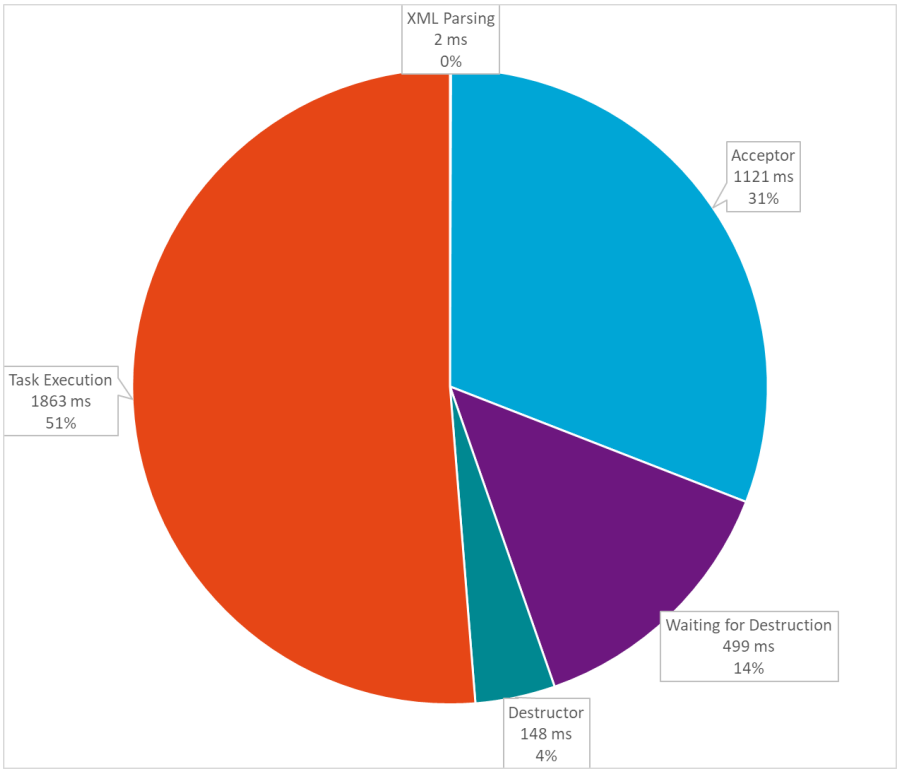


Figure 7.2: The average completion time when running one hundred no operations tasks. These tasks were executed on one machine with two CPU cores with twenty task slots available

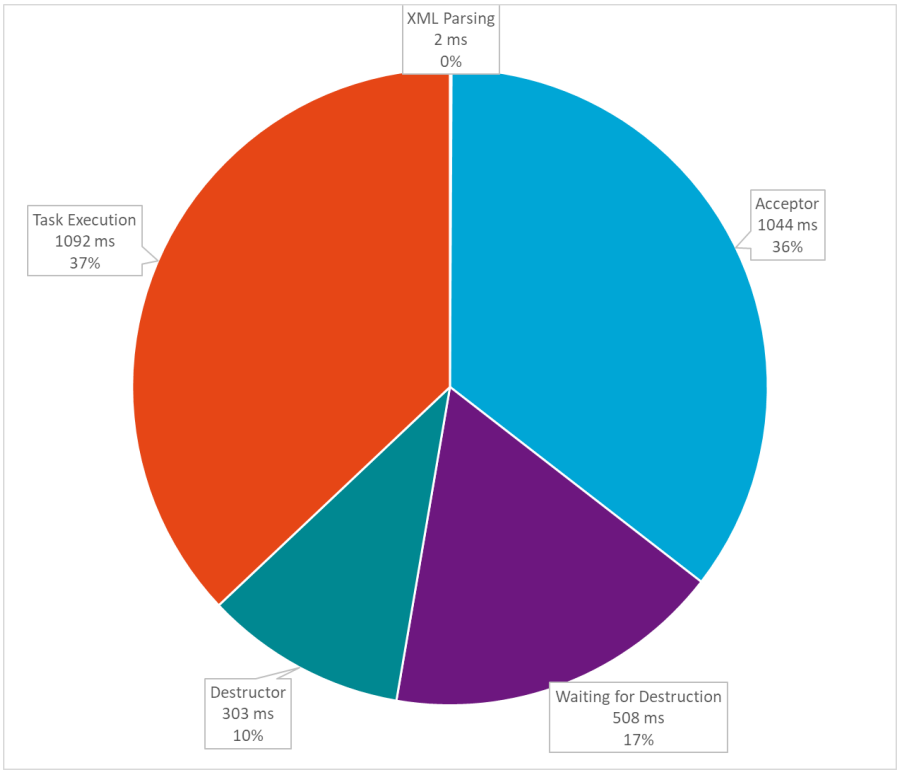


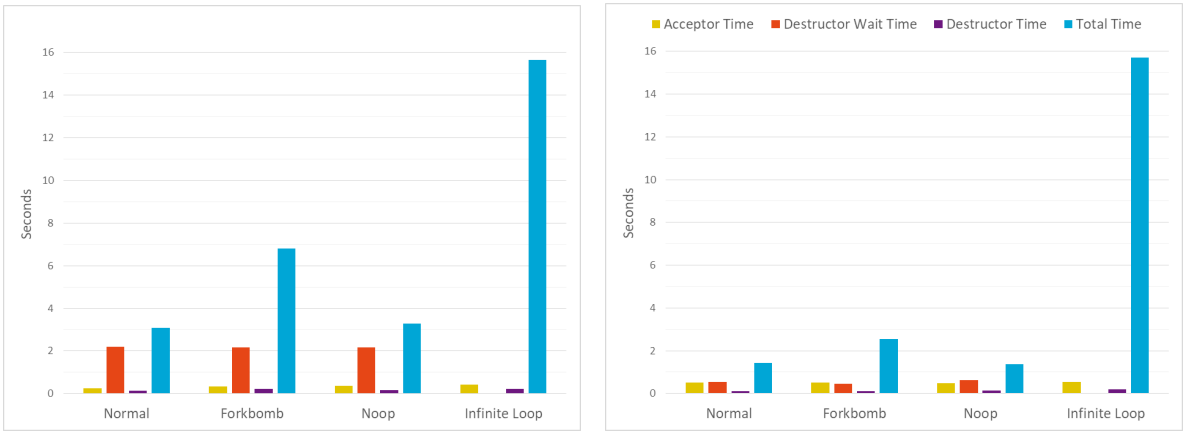
Figure 7.3: The average completion time when running two machines on one CPU core with ten task slots available each

As can be seen, the time spent in our system did not change much when comparing one machine to two machines. However, the task execution itself was much faster when using two machines with one core each.

The task execution time is the time that needed by Docker to execute the given task and can include some setup time. The much higher execution time was measured while executing the exact same test suite. Therefore we suspect that the difference is caused by an overburdening of Docker which increased the additional execution time. This could be because all tasks completed quickly and Docker, therefore, endured a high load. By dividing the tasks over two machines, both Docker instances have less work to do.

We decided to run the same test using normal tasks instead of no operations tasks. The results were very similar to using no operations. For a single machine, the normal tasks were even slightly faster (100ms) than the no operations tasks. This seems to support our suspicion that Docker is overburdened by all the tasks that complete so quickly.

As tasks in WebLab range from very short tasks (e.g. invalid code) to very long tasks (e.g. infinite loops), the optimal machine configuration needs to balance between that. We decided to test this by mixing all types of tasks together and by again running a hundred tasks on both configurations. Some of these tasks contained no operations, some contained a correct implementation, some contained fork bombs, and some contained infinite loops. The average times of all of these tasks, broken into the main four parts of their run, can be found in Figure 7.4.



(a) Tasks ran on one machine with twenty task slots, using two CPU cores

(b) Tasks ran on two machines with ten task slots, each using one CPU core

Figure 7.4: The completion time when running a mix of all types of tasks on two different machine configurations

As can be seen from the figure, the execution times as well as the destruction times are much longer when using one dual core machine compared to using two single core machines. This seems to suggest that multiple less powerful machines are better than a single very powerful machine. However, the results should be taken with a grain of salt. Results might be different when using non-virtual machines instead of virtual machines.

What we certainly can interpret from all these tests is that our system remains stable regardless of the types of tasks. Forkbombs and infinite loops are handled correctly and have only small impact on normal tasks.

8

Process

As this was one of the first projects of this scale we have worked on it is useful to reflect on the process. We will discuss our approach in section 8.1, and give a short reflection on this approach in section 8.2

8.1. Approach

We decided to use Scrum for the majority of the project. This was an easy decision to make, as we have used Scrum numerous times before in previous projects. Because GitHub has added support for "projects", which are basically Scrum boards, we created an issue for every task of each sprint. To have an easy overview of these issues a file was automatically generated which contained all tasks, together with descriptions, assignees, the estimated effort, and the priority. A preview of such a file can be seen in Figure 8.1.

User Story	Task	Member responsible for task	Task Assigned To	Estimated Effort per Task per person (in hours)	Priority (A-E) (A is highest)
#5 - Hardened Kernel	#6 - GRSecurity We need to communicate with grsecurity to see if we could obtain a license for the security patches.	Chiel	Chiel	2	C
	#7 - Applying patches We need to see if we can apply the grsecurity patches to our virtual system image.	Chiel	Chiel	10	D

Figure 8.1: An example of one of our task backlogs.

However, for certain parts of the project, we deemed Scrum would not be the optimal approach. For example, during the research phase, it would have been very difficult to divide tasks, as it would have been impossible to know exactly what would have had to be researched beforehand.

Another example would have been week four, where we decided a working product would have to be delivered at the end of the week. Because of this aim, certain parts would have had to be finished before others. Restrictions like these would be impossible using Scrum, without changing our sprint length to a single day. Because of this, we decided not to use Scrum when such an occasion arose, which only happened during these two weeks out of the entire project.

8.2. Reflection

Our decision to not use Scrum for certain weeks was, in our eyes, the right decision. Everything was still finished by the end of the week, but there was no time wasted on waiting for people to finish tasks. Everyone worked together correctly, and everything was finished in an orderly manner.

Even though this was the case, we are still happy that we used Scrum for the other weeks. Having a clear outline of what needs to be done every sprint and who needs to do it adds a lot of structure. Suddenly being fully responsible for what gets done, and when it gets done can be overwhelming. Following a scheduling structure we are all familiar with has helped us stay on track. Because of this, we were able to finish the project in a timely manner.

9 Ethics

In this chapter, we will discuss the three main ethical issues, which arose while creating the backend. In 9.1 we will discuss the privacy concerns which exist when using such a system. In 9.2 we will discuss if it is ethical for us to replace people with our automated test system. Lastly, in 9.3 we will discuss the ethics behind automatically flagging users as having cheated on an assignment.

9.1. Privacy

One of the main ethical concerns is the privacy of the users. As we have built a new backend for WebLab, we don't have our own frontend that users interact with. The backend receives tasks from the WebLab frontend. Each task contains the following information:

- the randomised unique identifier for the task
- the programming language
- the setting for reporting program output
- the name of the image to run the task with
- the status of the task
- the program written by the student
- the tests written by the student *or* the specification tests

None of these items can be used by us to identify a student. The only possible link to a certain student could be contained in the actual program and tests. If a student were to decide to include his or her name as a comment, we would have access to this information. However, these files never leave the system and are deleted upon completion of the run. Because of this the information is still safe, even if someone were to include identifying information.

Unfortunately, we have no explicit control over the machines on which these files are used. The machines are added to the system by an administrator, telling us we can use it to execute student code. Because of this, we can unfortunately not make any guarantees regarding the safety of these machines. But, if the system administrator follows our recommendation of using a hardened kernel we can say without a reasonable doubt that the user files are handled safely.

9.2. Replacing People

A different crucial point is the replacing of student assistants with our backend. We have created a way of automatically testing and grading the assignments handed in by students. If all assignments were currently graded by people, they would be out of a job. However, the current backend implementation for WebLab already uses automated testing. The only addition our system makes is allowing languages to be tested which do not run on the JVM. Because of this, we believe our system is ethical. We are not replacing humans with software, as there were never humans doing this job before.

9.3. Automated Flagging

If we have determined a student has attempted to cheat, we add a flag to their assignment saying they have tampered with their results. This boils down to the fact that we label people as potential cheaters, even if no human has verified this claim. Our tamper flag is meant to signal a course instructor or student assistant to take a closer look at the assignment. If they deem the user did indeed tamper with the result, they should take the appropriate actions. It is not our intention to have people fail a course simply because they got flagged. Our intent is for graders to have another tool to identify the assignments they should take extra time on. Because of this, we believe it is ethical to have such an automated flagging system. No drastic measures should be taken simply because a user cheated according to our system. People should still identify if the student actually cheated.

10

Discussion and Recommendations

The backend is designed to allow a large number of users to run and test Python code in a safe environment. We also aimed to create this system in a flexible manner, in that it can easily be expanded with new features. This chapter takes a look at possible new features that could be added in the future. For these features the design implications are discussed, including possible solutions and recommendations for the implementation. First image management is discussed in section 10.1. This is followed by the implications of implementing an interactive console for communication between the user and their task in section 10.2. Section 10.3 elaborates on the possibilities and dangers of allowing more than simple text as output. Then a note is made in section 10.4 on how our system can be scaled beyond its current limits. Lastly, section 10.5 discusses how image building and deployment could be implemented.

10.1. Extra Languages

As our backend is based on Docker images that run the user tasks, new languages can be added by providing new images. Creating those images is simple (as can be seen in F.1), but requires some attention as a badly created images may have effects on the performance of the system. First of all, the created images need to be stored on or transferred to the servers running the tasks. Big images fill the disks of the servers and use more bandwidth when transferred. It is, therefore, best to create images that are as small as possible and do not include any features that are not needed for running the task. As removing an image from the set of created images renders any course that uses this image useless, an image should only be removed with caution. Rebuilding an image when it is needed seems to be a solution for this, but as this can take a few minutes, this method is not very useful. When images can be added more easily than they can be removed, there is the danger that the number of images will rise steadily until the problems mentioned before start to occur. Therefore, new images should be added with care. This is also why we dropped the option for each assignment to define its own image and replaced this by a global set of images to choose from in the backend. This way each supported language should only have a couple of images that can be used by the courses, for example a minimal install and some with a specific set of libraries installed.

10.2. Interactive Console

A possible expansion of our system is to allow stateful connections between the user and the process running in the container. The current system is not built to directly support this type of connection.

First of all, this requires there to be some direct connection between the container and the user. This connection can be made using socket connections but requires some additional steps. As the connection must be made from the user's browser, web-sockets [24] should be used. Making a direct connection between the user and the container would be a very bad idea, as this exposes the machine IP address to the user. It makes the machine a potential target for DDOS attacks, malicious connections and potential hacks. Therefore, the backend or frontend would need to relay the connection instead. This means that the user has a stream connection with the backend or frontend, which in turn has a stream connection with the container.

While additional connections are not necessarily a problem, it would require significant modifications to both ends. Currently, the backend and frontend are completely separate. The only interaction between them

happens via task files. This makes a stream connection between them a lot more difficult. The frontend and backend should use a different way of communicating when streams should be supported.

Another change that is needed to support this feature is caused by the resource limiting done by the backend. As we limit on execution time, a connection would be closed automatically after a certain amount of time. With interactive connections, containers should be kept alive until it is determined that the user is no longer actively using the container. This, in turn, would require certain mechanisms to be disabled or changed in the backend, e.g. the stale container checker. The frontend would also need to properly close the connection to ensure that containers are removed correctly.

Lastly, this interactive mode requires many changes to how the testing aspect of the assignments works. Currently, tests are run on the solution file, but this file is not necessarily present when running interactively. The tests can be based on an output file that needs to be present at some point or at the commands used to perform certain tasks. The tests can even be disabled and replaced by some manual process.

These changes have effects on many aspects of the current system. For example, when using this interactive mode for graded assignments, the state of a user needs to be stored. You can either store this as a list of all used commands or as a full copy of the container. The first has a greater initial starting time as the commands should be rerun, while the second option uses way more disk space. These solutions can both be problematic at some point. For graphics-processing assignments, this rerunning of commands can take a lot of time. It provides a way to launch a denial of service attack on the system by continuous relogging. Storing the full machine state for every assignment of every user becomes a dataset that quickly grows in size.

From this, we can conclude that implementing this feature in our system is possible but requires some big design changes and decisions to be made. The effects of these decisions can have big impacts on the usability of the feature, the system performance, and system security. Therefore, extreme caution is needed when implementing the interactive mode. As such, we recommend not to add such a functionality to WebLab.

10.3. Advanced Output

Currently, WebLab only supports textual output from the tasks. The new backend, however, supports any output of the user as the output is fully file-based. Additional files that are created by the user are placed in the output folder of each task, but currently, only the test-result file, the `stdout` and the `stderr` are parsed and sent back to the front-end. Enabling the additional output formats therefore only requires some small changes in the backend. For the front-end, more work is needed as there needs to be a way to present this output to the user. It is not advisory to just put the output on the web page as this creates security problems. It can, for example, be used to inject HTML and JavaScript into the result page and therefore enables cross-site scripting attacks. A malicious user can also use this technique to steal the session cookies of a TA, as they will see the output generated by the original author of a submission. This can be partially fixed by blocking/escaping all the JavaScript code, but this renders (Python) libraries like bokeh useless as they use HTML and JavaScript to draw graphs. Embedding the results in a separate frame on the page also does not fix this problem due to them being on the same domain [6]. Even in a new window, the cookies can be reached as long as the domain is the same. Therefore the output must be shown in a separate window on a different domain. This way the Same Origin Policy (SOP) ensures that, among others, the cookie data is safe [6]. But this, sadly, does only work for HTML/JavaScript output. Images can contain executable code itself [42], so a TA might need to be cautious when looking at the solutions when these consist of graphical output. But to be fair, this risk is present on any website. You can also make the results downloadable and thereby provide support for all types of output files. This again boils down to the question of how secure it is to download random files from possibly malicious users, who were able to hand-craft those downloads and could have, for example, packaged a virus in them.

Any output besides text is not totally risk-free from a security perspective. Downloading the results would be the worst solution. Limiting the output to images, text and HTML/JavaScript, seems to be the best solution. These forms of output can be made reasonably safe while being flexible enough to be used in many use-cases.

10.4. Scalability Improvements

Although our system is built to scale with demands, this scaling has its limits. Therefore, there could come a point where the backend becomes the bottleneck in the execution of tasks. As the backend is based on the idea that every server is capable of running all the tasks, this means that every server must be equipped with every image that could be needed for running a task. For a system with such high loads, specialised servers only capable of executing a single type of task might be preferred. As implementing this in the backend

introduces many problems with thread sleeping/waking (see subsection 3.10.2), we decided on the current implementation. Improving the code to support this is therefore not simple. It can, however, be done in a simple way that is similar to how we changed the frontend to split the JVM tasks from the Docker tasks (see section 6.2). The backend can be started multiple times, each listening in a different folder for different types of tasks. The WebLab frontend must then divide the tasks accordingly. It would then be possible to run multiple specialised instances, just like the JVM-based backends.

10.5. Image Deployment

In its current form, the backend can only work with machines which already have all the needed images available on it. It would be much easier if images could automatically be distributed to all machines. Whenever the information for a new machine is added, all missing images should be transferred to the machine. The files to build an image could be added to a specific location. After adding the correct information to the settings, the image could then be built on either a locally running virtual machine, or on one of the available machines.

Care should be taken that the stale container checker does not remove the container that is building the image. The machine should ideally be removed from the system temporarily, to ensure that the image is built without problems. This is why a virtual machine running locally could be useful, as it would make the image building process easier.

11 Conclusion

Over the last three months, we have designed and created a new backend for WebLab. We have researched the possible alternatives to a completely new backend and looked into how to safely execute unsafe code. The new backend adds supports non-JVM languages, which is a feature multiple course instructors requested.

All needed features have been implemented, and the system has been thoroughly tested under synthetic load. Parameter tweaking will be necessary to be on par with the current backend, but this can be done when and if the backend is rolled out.

Creating a service which people rely on is a stressful experience, cases one normally would not take into account suddenly become extremely important. Occasional crashes could result in exams having to be cancelled, which is a scenario that should never happen. Because of this, we have gained the experience of working with a well-used system and have realised all the difficulties which arise when doing so.

We believe the backend is a suited solution for this project, given the limitations and time constraints. We would like to see, or even perform, further development. There are possible improvements to make the system even more robust and add features which were unfeasible in this time frame.

We would like to conclude that the backend is a solution we can be proud of creating. Because of the implemented features and the reliability of the system we deem the backend to be a success.

A

Software Improvement Group

As a tool to verify the quality and maintainability of our code, we needed to upload our code to the Software Improvement Group (SIG) for review. As we will receive the response on the second submission after the deadline for the report, this chapter only lists their response on the first submission made. The changes that were made to fix the problems found in the submission are discussed in section A.1. Please note that Figure A.1 is the received response and therefore is in Dutch. In addition to the made changes after the first submission, section A.2 discusses the changes we made to improve our score before the second submission.

A.1. First Submission

The first submission was made roughly in the middle of the project on May 31st. The response received after this submission can be found in Figure A.1.

A.1.1. Changes Because of First Submission

The code duplication issues in `CustomLinkedBlockingQueue.java` were fixed by simply removing the unneeded functionality which introduced the duplicate code.

The Unit Complexity issues were resolved by splitting up the methods in the TimeoutChecker, and by refactoring the rest of the code base in a similar way.

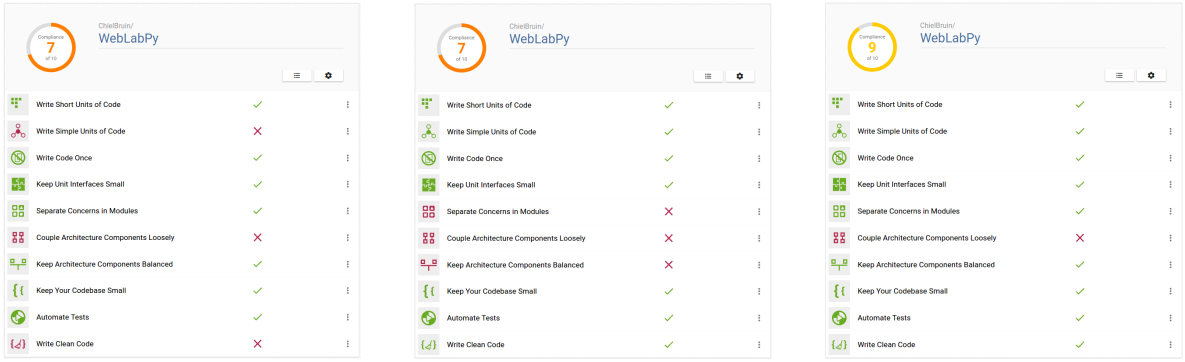
```
[Analyse]
De code van het systeem scoort 3 sterren op ons onderhoudbaarheidsmodel, wat betekent
dat de code gemiddeld onderhoudbaar is. De hoogste score is niet behaald door een
lagere score voor Duplication en Unit Complexity.

Voor Duplication wordt er gekeken naar het percentage van de code welke redundant is,
oftewel de code die meerdere keren in het systeem voorkomt en in principe verwijderd
zou kunnen worden. Vanuit het oogpunt van onderhoudbaarheid is het wenselijk om een
laag percentage redundantie te hebben omdat aanpassingen aan deze stukken code
doorgaans op meerdere plaatsen moet gebeuren.

In dit systeem is er bijvoorbeeld duplicatie te vinden in
CustomLinkedBlockingQueue.java, zowel tussen de twee poll-methodes als tussen de
take-methods. Het gemeenschappelijke gedrag wordt alleen binnen deze class gebruikt,
en kan dus makkelijk generiek gemaakt worden. Het is aan te raden om dit soort
duplicaten op te sporen en te verwijderen.

Voor Unit Complexity wordt er gekeken naar het percentage code dat bovengemiddeld
complex is. Het opsplitsen van dit soort methodes in kleinere stukken zorgt ervoor
dat elk onderdeel makkelijker te begrijpen, makkelijker te testen is en daardoor
eenvoudiger te onderhouden wordt.

In jullie geval zijn sommige methodes onnodig complex omdat ze te veel
verantwoordelijkheden hebben. De class TimeoutChecker is hier een goed voorbeeld van.
Alle logica staat nu in één grote methode, run. De complexiteit kan worden verlaagd
door het probleem dat in deze methode wordt opgelost op te splitsen in deelproblemen.
De complexe boolean conditie (task.done || task.fail || task.timeout || (start =
task.startTime) == 0L) kan bijvoorbeeld naar een nieuwe methode "isAlreadyHandled"
worden verplaatst.

De aanwezigheid van test-code is in ieder geval veelbelovend, hopelijk zal het volume
van de test-code ook groeien op het moment dat er nieuwe functionaliteit toegevoegd
wordt.

Over het algemeen scoort de code dus gemiddeld, hopelijk lukt het om dit niveau nog
wat te laten stijgen tijdens de rest van de ontwikkelfase.
```

Figure A.1: The received response from the first SIG submission

A.2. Second Submission

The second submission was made at the end of the project on June 23rd. Due to the fact that the analysis of the submission takes approximately one week, the results of this submission were added after the deadline for completeness sake. To determine the code components that needed refactoring to improve the code quality before this submission, we used another tool created by SIG to ensure the results would be comparable. This tool, called Better Code Hub[1], analyses the code and gives a score on ten criteria with a suggestion for changes that can be made to improve the score. The scores at the time of the first submission can be found in A.2a.

[1] `https://bettercodehub.com/`



(a) The score at the time of the first submission

(b) The score before final refactoring

(c) The score at the time of the second submission

Figure A.2: The Better Code Hub scores achieved during the project

In A.2b the scores can be seen right before we started our final round of refactoring. By comparing these to the first results, we can see that the complexity of our code has gone down and that the code became much more clean. This made that the corresponding metrics become green, but due to the growth of our system two other metrics became points of attention. The balance of the components was improved by refactoring the package structure and removing old and unused classes and methods from the project. To reduce the separation of concerns in modules, responsibilities of classes were cleaned up. An example of this is moving the starting of a task on a container to the `Task` -class instead of the `TaskManager` . Besides these changes we also went over all the `\\TODO` -tags in our code and either removed them when they were no longer needed, implemented them or decided that they were future improvements and that they should be left untouched. With these changes we were able to increase our score on Better Code Hub to the one displayed in A.2c.

The results of the second submission are included in Figure A.3. SIG acknowledges that we have addressed the outlined problems and that the maintainability score of our code has improved for the second submission.

```
[Hermeting]

In de tweede upload zien we dat zowel de omvang van het systeem als de score voor
onderhoudbaarheid is gestegen. We zien bij zowel Duplication als Unit Complexity een
duidelijke en structurele verbetering, waardoor jullie ook een verbetering in de
totaalscore hebben gerealiseerd.

Ook is het goed om te zien dat jullie naast nieuwe productiecode ook aandacht hebben
besteed aan het schrijven van nieuwe testcode. De hoeveelheid nieuwe tests zit er ook
erg goed uit.

Uit deze observaties kunnen we concluderen dat de aanbevelingen van de vorige
evaluatie zijn meegenomen in het ontwikkeltraject.
```

Figure A.3: The received response from the second SIG submission

B

Project Plan

B.1. Project Description

WebLab is a system where students can write and test code using an online interface. Course coordinators can create assignments and exams. They define tests which make sure the student code meets requirements.

In its current implementation WebLab does not support programming languages which do not run in the Java Virtual Machine (JVM). As certain courses are currently requesting support for Python, a solution has to be found to make this possible. The current Jython implementation does not support essential libraries requested by users, and has been determined to not be an adequate solution. To realise this solution we need to design a system which is safe to use and scalable. Because people should not have access to the operating system the code is ran on, access needs to be blocked. Other security risks such as creating a denial of service attack in WebLab or tampering with test results must also be impossible. The solution to this problem must also be able to be deployed when there are a large number of concurrent users, as WebLab is also used during exams.

B.1.1. Company description

The TU Delft programming languages group *"conducts research into concepts and techniques for programming language design and implementation. The Academic Workflow Engineering team of the PL group applies advanced programming languages techniques to the development of software to automate academic processes."* [21]

B.2. Design Goals

To guarantee a well working product will be delivered, certain goals will need to be reached. These goals are described in the following paragraphs.

Performance

The designed system should not use all resources on the server it is ran on. It has to be lightweight to support multiple concurrent users and not hold up other services on the server.

Security

As our product will execute code created by the user, it has to be able to cope with malicious code. Even if a user is able to compromise parts of the system, the influence of this must remain limited to the user themselves, and should not benefit the user with regard to their grading.

Maintainability

The code which is delivered at the end of this project will be used as the back-end for WebLab. As a result the code needs to be maintainable. To guarantee maintainability we have defined three metrics.

Code Quality The code must be of high quality. To ensure this, static analysis tools will be used. As it is currently undecided which programming language will be used, the exact tools are to be decided.

Testability The code must be testable and a high test coverage must be ensured. This assurance will result in less bugs and code which is easier to maintain.

Documentation To ensure our code will remain maintainable it will be well documented. This way anyone that is a (future) part of the development team maintains a clear overview of the functionality of all the code components and methods.

Scalability

As WebLab is also used during exams the solution we design must be scalable. First year courses currently have up to 500 people who all have to take an exam at the same time. If the system can not support a load this big, the system is inadequate. As a result we need to guarantee a stable solution which is scalable to support a high amount of concurrent users.

B.3. Requirements

We define our requirements with the MoSCoW model [26].

- Must haves
  - WebLab will be able to execute Python code
  - Python programs will be executed in a safe environment
  - Code written by the user can use Python native libraries
  - Code written by the user can use assignment defined Python libraries
  - A Python program of one user cannot influence programs of other users
  - Support for multiple concurrent users without user noticeable problems
  - The user must not be able to tamper with the test results
- Should haves
  - Support for a large number of (100+) concurrent users without user noticeable problems
  - Course specific configuration where the coordinator can choose the modules available to the user
  - Well designed job scheduling to reduce latency
  - Maximum CPU usage can be limited
  - Maximum memory usage can be limited
  - Soft crashes should be automatically resolved
- Could haves
  - Graphical output of python programs is shown on WebLab
  - Implemented support for other non-JVM languages such as C++
  - Interactive communication between WebLab and a simple input/output Python program
  - Interactive communication between WebLab and a Python REPL

B.4. Approach

The project will be divided in two distinct parts. The first part will focus on a literature survey on what is needed to implement the features described above. We will create a design for the system in this phase. In the second phase, we will implement the designed system. SCRUM will be used in this part of the project [5, 40]. SCRUM cycles will be one week long. Daily meetings will be held to ensure a smooth development process. Weekly meetings will be held with our coach.

Development team

The development team exists of the following people:
Taico Aerts
Chiel Bruin
Bram Crielaard

Scrum

We have assigned the different scrum roles as follows:

Scrum master	Taico Aerts
Product owner	Bram Crielaard
Development team	Taico Aerts, Chiel Bruin, Bram Crielaard

Planning

The project is divided into three phases. The first phase is the research phase. During the research phase, we research possibilities, limitations, and similar products based on which we create an initial design for the system. At the end of the research phase we will deliver a research report, which will outline what we have found, an initial design for the system and why we have chosen for that design.

Following the research phase we start with the actual implementation of the system. We will first finish the must have requirements and implement should haves where possible. The implementation is done in scrum sprints each concluded by a meeting with the stakeholders. At these meetings we will show the progress that we have made and confirm that we are not straying from what the client wants. Close to the end of the implementation phase, we will test our system by conducting penetration tests. For this, we will ask different people to perform an assignment where they try to break free from the system.

The final phase consists of bug fixing, writing and preparing our presentation, and finishing the documentation. At the start of this phase a feature freeze will be put into effect. This will ensure high quality code is delivered and no new possibly broken code can be introduced.

A more precise planning with weekly deadlines is given in the following list:

Week	**Item**	**Deliverable**
Week 1 and 2	Research phase	Ten page research report, Project plan
Week 3, 4 and 5	All must haves implemented, start on should haves	Working implementation of must have features which can be used for penetration testing
Week 5	Mid project meeting	Working demonstration of must have features
Week 6 and 7	All should haves implemented, feature freeze	Working implementation of (most) should have features which can be used for penetration testing
Week 8 and 9	Bugfixing, optimizations, code quality improvements	Finished software product
Week 9	Finalize final report	Final report
Week 10	Finalize presentation	Presentation slides
Week 11	Presentation	Presentation

C

Research Report

In this appendix we will explain the research we have done to realise this project. We will outline nine different topics that we have researched, why they needed to be researched, and how our findings influenced design decisions. As key components have to be decided early on in the project, we will also explain how we derived the overall design from smaller sub-components. In this appendix we will try to determine how we can create a system in which we can safely run and test user code, without the system crashing or the user tampering with their results.

We will discuss how we can create a safe Linux environment in section C.1. In section C.2 known exploits in Python will be discussed. In section C.3 we discuss what ways exist to sandbox Python itself. We discuss in what way course instructors should be able to configure courses in section C.4. In section C.5 we will explain how output can be displayed to the user. Section C.6 defines several ways of managing virtual machines. In section C.7 we discuss similar systems which are currently on the market, and in section C.8 we define a way we can safely test user code. Finally, from these questions we have created our initial designed, outlined in section C.9.

C.1. Sandboxing the User

There are many different ways for users with malicious intent to cause harm to the functioning of the entire server. Therefore the server also needs to be protected in many ways. In concept, the easiest way to provide this protection is to give each user a separate server. This way a user can completely kill the server on which their code is ran, without causing any effect for the other users. The requirement of a different physical server for each user can, however, be impractical with anything more that a couple of users. As our system must be deployed in a setting where multiple hundreds of people can use the system concurrently, such a solution is not feasible. We need to create a system that, preferably, can run on a single server. In order to achieve the ideal of each user having their own environment, we need to be able to create multiple fully walled off sections on the server. In these sections, the user can execute all the code they want, as they are not able to escape this sandbox they are placed in. In this section we will be discussing multiple ways of constructing such a strong sandbox for the users. First, the construction of such a sandbox using features of the Linux kernel can be found in subsection C.1.1. Subsection C.1.2 will then list different tools that simplify this process.

C.1.1. Kernel Protection

The Linux kernel provides many ways to monitor and limit running processes. Three different groups of measures are outlined in the following sections. First, we look at kernel functionalities that can limit the resources available for user processes. We then explore ways to deny a process from calling certain system functions. Finally, methods for hardening the kernel are discussed, which make it even harder to use exploits.

Managing resources Limiting the resource usage of a single user can be a very useful tool in restricting the negative effects that a user can have on other users. A user can, for example, run code that uses all of the CPU-resources of a system, which would prevent other users from using the CPU. Similar methods can be used to hog up server memory. Restricting CPU and memory usage is therefore required to ensure the availability of those resources to the other users on the system.

A simple method of restricting the CPU-usage is by setting the priority of the process to a lower value than other processes. This priority can be set using the Linux command `nice` [32]. This method does, however, have a big disadvantage which makes it unusable in our system. When assigning the same priority to all users, a user can still claim a big chunk of the CPU-time as there is nothing with a higher priority. In combination with the fact that the `nice` command can only be executed at the start of execution, the server would have to know beforehand which processes are CPU hungry. As this cannot be determined, usage of this function will not be feasible. A similar command which addresses this problem is `renice` [29]. This command can be used to apply priorities to processes after they were created. By using a script that monitors CPU usage of running processes we can now create a system that dynamically sets the priorities of processes to balance the CPU time they use. One case that is not solved by this method is a process that never terminates. Such a process can keep using its limited percentage of resources indefinitely. There are, however, more powerful commands built in the kernel that remove the need for such a management script and offer protection for long running processes.

One such commands is the `ulimit` [28] command. With `ulimit` we can limit both the CPU time and memory usage of a process and make it impossible for a single process to claim all the system resources indefinitely. This command also has a big disadvantage in that it can give users with malicious intent more system resources than should be assigned to them. The way a user can achieve this is by using multiple child processes. The `ulimit` command does not use a cumulative count over all the child processes and therefore you can enlarge your resource quotas by utilising forks in your program. This can be exploited easily by using a ForkBomb [34]. As this practice is well known, the command has a countermeasure built into it. It works by limiting the maximum number of child processes that can be created. As certain programming tasks require multiple child processes (for example the exercises that learn you the concepts of forking in C), we would rather limit the total usage of one user than limiting the amount of processes and the amount of resources per process. While `ulimit` could be used, it might not be the perfect solution in our case.

Another way provided by the Linux kernel to limit the resources of a process are `CGroups` [30]. This method provides the benefits of `ulimit` while also taking care of the child processes. Control groups are built in a hierarchical structure which allows to use such a control group within another CGroup. When this is done, a subgoup cannot exceed the limits imposed by its parents. This way we can use this system very effectively to divide system resources between the server management and the user space, while also doing the same within the user space for each user.

Besides CPU and memory usage, we should also restrict the files that processes have access to. With `chroot` we can do exactly that. It creates what is effectively a local filesystem. A user process will only see a subset of the actual filesystem, and cannot access anything outside of it. There are some disadvantages to the usage of this command that can limit its practicality in some applications. Firstly, caution is needed when creating the sub-filesystem. When the filesystem is not entered properly after creation, it is trivial to escape the `chroot` and cause harm to parts of the filesystem that should not be accessible [4]. Krohn et al. [25] adds to this that jails constructed using `chroot` are often more resource intensive as all files (and libraries) used by a process must be contained inside the jail. Maintaining these multiple copies of files can be administratively difficult [25]. Most notably, a chroot jail can easily be escaped by a process after it knows that it is placed in such a jail, making this method not very effective [46].

Managing Systemcalls Besides system resources that need managing, the operating system also enables features provided by system calls that need management when running user code. These system calls are used for various things including forking and file reading/writing. A malicious user can use these calls to perform many complex system operations and as an effect cause harm to the system. We therefore need a way to control the access a user has to those functions. Simply blocking all of them may render the operating system useless for many normal programs, so this is not a solution that fits our needs. A kernel function with which we can limit the access to system calls is `seccomp`. `seccomp` blocks all calls except `exit`, `sigreturn`, `read` and `write` [31]. But as already said, some additional system calls may be needed to teach concepts like forking. Luckily, there exists an extension of this function that provides filters that can be applied on the system calls. This function, `seccomp-bpf`, can therefore be used to create a restriction system that can give access to certain features only when needed.

Hardening the kernel All the methods listed in previous sections are based on built-in functions of the Linux kernel. This makes them as safe as the kernel itself. If the security baked in the kernel is improved, the

safety of the entire system will be improved as an effect. There are two main methods to improve the safety of the kernel, also called hardening of the kernel, that will be discussed here.

Grsecurity is a very strong hardening patch for the Linux kernel. A major component of grsecurity are the PAX kernel patches. These patches lay many restrictions on the usage of memory by processes. It, for example, flags data memory as non-executable and program memory as non-writable tot prevent, among others, buffer overflow attacks. It also implements address space layout randomisation (ASLR) as a way to prevent attacks based on simple to guess memory addresses. In addition to these security measures from the PAX patches, grsecurity adds one more major security improvement over the vanilla Linux kernels. These normal kernels have users that have, in its most simple form, a binary permission system with normal and superusers. Grsecurity uses role-based access control (RBAC) that offers a much more fine grained method to allow certain users to perform certain tasks [22]. This way a user that only needs to use one function that requires a superuser, does not need to get those elevated permissions across the board. It is also possible to create users that only have access to one single type of operations. This way it becomes even harder for restricted users to misuse their set of privileges to gain access to more parts of the system. A last important feature of grsecurity is that it restricts the functionality of `chroot` in a way that disables certain system calls from inside `chroot` and prevents user privilege escalations. This makes the usage of `chroot` safer.

A second hardening method is by using SELinux. This security module is based on Flask, a mandatory access control (MAC) implementation by the NSA. While this tool is, in contrast to grsecurity, embedded in the mainline for many Linux distributions it does not offer such a complete set of protection measures. SELinux is mainly focused on providing a MAC implementation [36] similar to the RBAC sytem implemented in grsecurity. This implementation in SELinux is stronger than the one in GRSeurity [17] which makes this tool more favourable. In addition SELinux is freely available, while grsecurity can only be used commercially. Grsecurity is however the better system as it is easier to use and also includes many more protective features [17].

C.1.2. Methods for Sandboxing

With methods described in subsection C.1.1 it is possible to create a sandboxed environment on a machine in which code can be executed. A program that uses many of those methods to create sandboxed environments is Docker [8]. Docker provides a simpler interface than the raw calls to construct the sandbox. It is also open source and very popular, resulting in active development. As an effect it is probably safer to use Docker than to create such a system from scratch. Although an environment using docker may not be perfectly safe [23], it is hard for a non-superuser to break out of the sandbox and reach the main machine, creating an effective barrier.

Such a barrier will, however, never be as safe as running the code on a virtual machine, as this separates the host and the VM to a much higher degree than containers can achieve [35]. A virtual machine has a few disadvantages that make it less useful for quick deployment and less scaleable to hundreds of users. This is because a container based solution can reuse many parts of the hosts operating system and can therefore be more lightweight than a VM that needs to include all the files necessary to function as an operating system. This means that VMs have a higher memory (and disk space) usage and are relatively slow to deploy. When this overhead is multiplied by numerous users, the resource demand is significantly higher than a container based solution. A best of both worlds approach is to run multiple users on a single VM, using the VM as a shield between the containers and the server[35]. In this case the overhead is spread over multiple users which makes the system more scaleable.

C.2. Exploits in Python

Python is a language which is often used in hackathons. This results in many people taking it upon themselves to break free from the "jails" which are provided by the organisers. These exploits are rather ingenious and display a certain wit. Because of this, hackers often enjoy sharing and explaining the code they used to break free from their restricted environment. Because we are designing such an environment ourselves it is beneficial to research these exploits. Preferably, we would even implement a system to prevent such attacks from happening in our environment.

In this section we will discuss the different aspects which go into finding and designing such an exploit. First of all, we will explain how easy it could be to perform such an attack in subsection C.2.1. Secondly, we will analyse if it would be feasible for a user to perform such an attack in subsection C.2.2. Last of all, we will determine if these types of exploits are a valid concern in subsection C.2.3.

C.2.1. Ease of attack

Attacking can be relatively easy, but does requires a certain amount of knowledge about the system. An exploit which is very relevant to our research was found by Pike [37], who found a memory leak in the Python module NumPy. NumPy is one of the main reasons the TU Delft is looking into a Python interpreter which does not run in the JVM, as NumPy does not work in the JVM. Looking on GitHub for issues containing the string "segmentation fault" or "segfault" resulted in Pike finding many bug reports explaining which functions result in these errors. As many third-party Python modules are written in C one can examine the C code, and hopefuly find a memory leak which can be exploited. This is exactly what Pike did, resulting in him breaking from from the sandbox which was provided. Because of this he was free to run any code which was not blocked on an OS level.

A different, easy to fix exploit, was explained very clearly by Batchelder [1]. His methods rely on using the Python function `eval`. This function takes a `string` and evaluates it as Python code. Let's say we were to use a system which simply uses `eval` with no globals set. If a user passes an import statement to `eval`, it will be evaluated and executed. A security system which only uses this one layer of protection can be easily circumvented. This, unfortunately, would not be the only way this exploit could occur. If we were to simply block people from using certain modules in their python scripts they could still use them with `eval`. This works because it checks with the interpreter if it is alright to import modules. It does not verify this with the user environment if someone passes a globals and locals variable[16]. As people can find the `__builtins__` even with an `eval` environment which does not contain them[2] they can pass this to gain an unrestricted environment. Blocking modules on an interpreter level would fix this exploit.

Reading about failed projects such as pysandbox shows us there are many more exploits that have not been covered in the previous paragraphs[14]. As a result this page could contain an almost limitless amount of examples, but will be limited to the two given, as they are both very relevant this project.

C.2.2. Feasibility of an attack

Both exploits which we have described in the previous section rely on the user either knowing a lot about the system, a lot about Python, or both.

The first attack can only be performed if the user knows about existing bugs in available modules, and has the source code for these modules. Not only would the user need to make assumptions about which modules are available before an exam, they would also need to make assumptions about the exact versions of the modules which are available. The NumPy exploit explained in the previous section only worked in a single release. There is no way for a user to access additional information such as known exploits for a particular version of a module during an exam. This would mean that the user would either have to spend a lot of time during an exam using trial and error to find these exploits, or learn exploits for numerous modules and versions by heart beforehand. As a result, we conclude an exploit such as this one is unfeasible for a user to perform during an exam. It would simply take too much time for a user to design an exploit during an exam. Or the user would need to have an incredible amount of luck, knowing the exact modules and frameworks available for every exercise. Of course the user *does* have access to this information during an exercise. However, as the user also has access to numerous helplines while performing these exercises this risk is negligible.

The second attack can be performed if the user has relatively in-depth knowledge about Python and knows how the server is implemented. As this exploit can entirely be negated by our interpreter configuration, this exploit is only feasible with a broken server setup. As a result we conclude this type of attack is also unfeasible.

C.2.3. Results of breaking free

It is important to determine what the user might gain from breaking free from our restricted environment. If we determine there is no way the user can benefit from such an attack (for example, change their test score) it might not even be worthwhile to design a system to stop these attacks. As it is virtually impossible for us to stop a user from breaking out of a Python sandbox, all security will need to be implemented at the OS level. As a result, the risk is only as large as we design it to be with our virtual machine implementation. Because of this, we determine breaking out of a semi restricted Python interpreter *should* be safe and not result in a massive security breach.

C.3. Python Security

After identifying possible exploits in the Python language, we also needed to look into ways of improving the safety of this language. This section therefore discusses different security measures that are specific to Python, as well as the feasibility of these measures for our system. Subsection C.3.1 investigates ways to limit the possibilities of the user in the language and methods described in subsection C.3.2 aim to sandbox the entire execution instead.

C.3.1. Language-Level Sandboxing

There is also the possibility of limiting the capabilities that the user has from within the language, by blocking access to certain language features. The pysandbox project is a program that offered such functionality, but it was discontinued because it was very difficult to do anything complex in the sandbox. Even very basic python code would be denied as that code could also be used as an exploit. The author of the project published that *"pysandbox is broken by design"* [14, 44]. Python itself just contains tons of ways to work around any protections, which means that a lot of things need to be blocked. But blocking everything makes it very difficult to do anything complex. Instead, the author advises to put the whole Python process in an external sandbox.

RestrictedPython, another Python sandboxing tool, uses a restricted compiler to compile and run user code with a restricted set of language features. It is not widely used, still in development and currently not extensively documented. We have been unable to get RestrictedPython to work with any Python code that uses a built in function (not even print). Therefore, RestrictedPython in its current state is not usable.

C.3.2. External Sandboxing

A few Python implementations support a form of sandboxing that limits a process from the outside. For example, PyPy offers sandboxing that is similar to OS-level sandboxing [39]. All input and output of the program, including any library and system calls, are sent through system out and system in and need to be handled by an outer process. It also offers functionality to set limits on the amount of RAM and CPU time used.

However, with sandboxed PyPy, only pure Python modules can be used [38]. Many popular libraries like NumPy and SciPy require quite a few functions that are implemented in C libraries, and would thus not be supported at all with sandboxed PyPy. Using the PyPy sandbox would limit the usability of the system significantly, to a level that is very similar to what can already be achieved with Jython.

C.4. Limiting on a Course Level

Not every course requires the same modules to be available to students. Courses such as the "Operating Systems" course require people to spawn threads. Most courses do not need access to this functionality. One could even go as far as to say that other courses *should* not be able to use this functionality. Allowing users to use `fork` brings a lot of safety concerns, which should be negated where possible. Furthermore, for certain exercises the function to implement might be available in libraries, making the possibility to import them unwanted. Because of this, course instructors need a way of selecting modules which are or are not available to students.

We will explain the two aspects which need to be analysed for such a system to be created. First of all in subsection C.4.1 we will look into an easy ability for course instructors to delete modules. Second, we will look for an easy system for instructors to manage modules for users in subsection C.4.2.

C.4.1. Deleting modules

The easiest way to block access to a module is by simply telling the interpreter it does not exist. This can be done by editing `sys.modules` to return `None` when trying to import the requested module. This method can be done by injecting `sys.modules['moduleName'] = None` at the top of every file a user submits. However, there are numerous ways around this block. As described in subsection C.2.1 people have managed to find all builtin modules and import them. This will still be possible with this type of block.

We have two options if we want to guarantee people can not use modules which they are not allowed to. The first is completely removing the module from the OS. This only works for modules on which no other modules depend, such as `pdb`. The other way is for us to create custom versions of these modules. These modules would contain edited versions of all methods we do not want users to access, making them unusable

for exploits.

C.4.2. Managing modules

Course instructors should be able to set up our system with relative ease. Because of this, a configuration system needs to be created which allows instructors to toggle modules on and off. This way environments are created for each course that only have the selected modules available for the students. This system should completely remove unwanted modules from the virtual machine, as they could possibly be accessed if they are still available.

C.5. Graphical Output

One of the intended use cases of Python in WebLab is to use it for data visualisation. For any kind of visualisation, a graphical user interface is greatly preferable over a test based interface. Currently WebLab does not offer any output other than textual output. This section will explore the possibilities of showing graphical output on WebLab for Python programs, as well as limitations and dangers that come with this functionality.

C.5.1. Images

Possibly the simplest of the different graphical outputs are images. If after running the user code, one or more image files are written to the output directory, these could be sent to the user for display on the web interface. An image is, just like normal text, simply data that needs to be sent to the web interface. It would require relatively simple changes to both the WebLab front-end and back-end to be able to display images in addition to console output.

Dangers and Limitations If an image is very large, sending it to the user could potentially cause problems for the system. To reduce bandwidth usage a file size limit should be added. Formats like JPEG, PNG and GIF are most suited for compact image sizes on the web and are supported by all browsers, so these should be preferred [19]. Optionally, images could be compressed to further reduce network usage, if this proves to be necessary. This would only be the case if network traffic becomes a larger bottleneck than the availability of processing time or memory.

C.5.2. Web Pages

Data visualisation libraries like Bokeh output their graphs as interactive HTML web pages. Supporting these web pages in WebLab seems trivial, as they can be displayed by browsers as-is. The only problem is that the user would be able to write arbitrary HTML and JavaScript code. When the user itself runs their program, they are the only one who see the results. As browsers already allow users to run arbitrary JavaScript code on any website, this does not provide the user with any additional capabilities.

The problems start when someone that is not the user itself, attempts to display the generated web page, e.g. a reviewer that is manually reviewing student code. For this reviewer, displaying the output means that the arbitrary code written by the user is now executed on the WebLab domain. This could possibly allow for cross site scripting attacks. Since WebLab only uses cookies with the `HTTPOnly` flag, user sessions cannot be 'stolen'. The problem is that the JavaScript code, if executed under the same domain name as the WebLab site, can execute any function of the main WebLab interface as the logged in user (e.g. the reviewer). These functions range from signing out, to grading and even deleting student solutions. The easiest way to prevent such attacks, would be to host these pages on a different (sub)domain entirely. Creating a temporary link on a different domain where this web page can be viewed suffices in preventing an attack through WebLab itself. By hosting it on a different domain, the capabilities of the web page will be the same as any other website on the internet, and thus be just as safe as any other website.

C.5.3. Arbitrary File Types

If we already support sending both images and web pages, it would be relatively easy to create a system that is able to support any kind of file. We can now no longer make any guarantees about the safety of such files, but we can allow the user to download such files from a different (sub)domain. While users should be informed that the files may be unsafe, such a system would increase the usability of WebLab.

To prevent users from using WebLab as a file sharing service, we would have to limit how large files can be and how long they will be made available. Storing only the files created during the last run of a program by a specific user should suffice. If the user needs to obtain the files again, they can simply run their program

again. WebLab already offers a revision history for the code, so previous versions of the code can be restored. As such, deleting files after a day or less would be fine for the purposes of WebLab.

C.6. User Instance Management

With a system that uses multiple VM's and containers, there must be ways for the system to manage those. This manager must be able to start and stop instances when users join or leave the website, detect instances that get broken due to (malicious) code execution and gather the outputs of the programs ran to send these to the corresponding users. The following sections will discuss the ways in which these features, needed for a good instance management system, can be implemented for VMs and Docker containers. After this it is still necessary to investigate, on a conceptual level, how the scheduling should be implemented using the discussed features, which is done in subsection C.6.4.

C.6.1. Startup and Teardown

In order to be able to manage the running containers we need to be able to start and stop instances and thus we need an overview of the running containers. For Docker containers there are built-in functions for creating, stopping or even killing containers. It can also list the running containers [9]. This makes management of these relatively simple.

These functions are not available by default on virtual machines, but tools exist for this management of VMs. Most of those tools make use of libvirt. This package provides all the desired functionalities for the management of VMs. These are accessible from multiple languages using their APIs [48]. A tool that stands out is Docker Machine. This tool is built to be used with VMs containing multiple docker instances. As an effect it implements direct communications to those docker instances. It can also be configured to run VMs on different machines, a feature that can be very useful when deploying the final system over multiple servers to support larger numbers of users. A disadvantage of this tool is that there are no APIs available and therefore that a bash script must be used to manage the VMs [7].

C.6.2. Garbage Collecting

Although a user instance should always terminate after finishing its execution or after a maximum number of execution steps, it might be possible for a user to break the container in such a way that this does no longer happen. Therefore there must be a way for the server to detect such broken processes and stop their execution. Docker, libvirt and Docker machine all provide the techniques to inspect the running instances [10, 13, 48]. It is however difficult to detect if a VM is compromised, therefore a process that routinely resets the running VMs can be used to limit the lifetime of any compromised instances.

C.6.3. Communication

For the server there needs to be a way to run code on the user instances and gather the results from these executions. To achieve this, there must be some communication between the instances and the main server. The most versatile option would be to create a TCP connection or a socket based system like VMCI [47], over which information is transferred. With this option we have to enable more network features, which would make the container less secure. At the bare minimum, the only network service available should be the SSH connection used to start processes in the container. This SSH connection is enough to provide the user code to the container and run all the tests. Retrieving test results can be a little more challenging. A textual result can be retrieved using the standard output from the SSH connection, but for larger files this method is not ideal. This method is not preferred for these bigger file as ssh is not intended for this usecase. For the transfer of entire files to the host `scp` should be used [45]. `scp` is designed for exactly this purpose and works using SSH. A completely different solution would be to export the entire contents of the Docker instances using `docker export` to a `.tar` archive. This would work for our system, but as not all of the files in the archive are needed for our output, this can introduce large overheads. A last option would be to mount directories of the host in the containers and place output files in those directories. With this method caution is needed as a wrong mount can make the system more vulnerable to attacks.

C.6.4. Scheduling

Virtual machines generally have a slow deploy time as it takes significant time to boot a VM [33]. Therefore a system that uses virtual machines as a protection layer between a server and the user, needs to have a scheduling mechanism that ensures the user will never experience this startup delay. This mechanism there-

fore has to start new VMs in advance. Just reserving a fixed number of machines for joining users has one major drawback. On low system loads it will be very unlikely that many users join at once and claim all those prepared machines, but during an exam, when over a hundred people attend, the likeliness of this is much higher. Therefore the amount of prepared VMs should be based on the current system load via a linear equation like $n = \frac{s}{f} + B$ with n the required number of VMs, s the number of users and f a load factor between 0 and 1. This ensures that all the users can never claim more that $f \times 100\%$ of the available VM space and that there will always be room available for new users to join. In the case of expected high system loads (exams, practicals or deadlines), the system must always be prepared to handle these loads. The equation will also not prepare enough machines on low system loads. Therefore a better equation would be $n = max(E, \frac{s}{f} + B)$ where E is the expected load and B is a baseline number of free machines on low loads.

A way of reducing the VM deployment overhead in general is to use a VM for multiple users at a time. This way, instead of starting multiple VMs for multiple users, a single machine could suffice cutting the accumulative startup time. This makes the previously mentioned formula for the desired number of virtual machines: $n = \lceil max(\frac{E}{k}, \frac{s}{f \times k} + B) \rceil$ with k the maximum number of users on a single VM.

C.7. Current Similar Implementations

Numerous systems exist which allow users to run code online. We have examined four of these systems to see how secure they are. We will discuss Jupyter in subsection C.7.1, Trinket in subsection C.7.2, ideone in subsection C.7.3, and Try It Online in subsection C.7.4. We will discuss the relevance of this research in subsection C.7.5

C.7.1. Jupyter

Jupyter[1] is an interactive interface for teaching programming languages. As it is meant to be installed on the computer belong to the user, the online interface is not meant to be used intensively. We first tried to get access to the bash shell using `os.system('command')`. For this we need to import the `os` module. This worked, and after further probing we concluded we were able to import every module we tried. After we had access to the shell we tried to simply delete the filesystem. This crashed our connection. We aren't sure if we actually deleted the filesystem or if we were kicked.

As this exploit broke the system for the user we tried to create a forkbomb. This crashed the kernel, the number of threads allowed being unlimited. Because of this we had to conclude the online interface was relatively unsafe. This, however, is not a problem for when the user has installed the software on their own computer. The only person they are harming by doing this once installed on their own computer is themselves. This does mean that it is not a suitable solution for our system.

C.7.2. Trinket

Trinket[2] is a website meant to allow users to share code. It is focussed on graphical output. We tried to import modules, finding none were found. Because of this we assume it runs the Python code through an interpreter written in javascript. Because of this it is unsuitable for our system, and no further research was performed.

C.7.3. ideone

Ideone[3] is an online compiler. We tried our usual tricks of deleting the filesystem. We determined we had access to all of the Python features and modules, but were limited by the operating system of the machine the code is run on. ForkBombs were killed after 5 seconds, resulting in us being unable to crash the system. Certain bash commands such as `cd` were blocked, but we were still able to inspect the filesystem using `ls`. We were, however, unable to write to the filesystem. This system appears to be similar to the one we are designing, as all blocking is done on an operating system level.

C.7.4. Try It Online

Try It Online[4] also is an online compiler. It is extremely similar to ideone, except for two key differences. First of all, the bash output is suppressed. This means we were unable to inspect the filesystem. The second dif-

[1] `http://jupyter.org/`
[2] `https://trinket.io/python`
[3] `http://ideone.com/`
[4] `https://tio.run/`

ference is the amount of threads we could spawn. We could find 116 different PIDs meaning we had spawned 115 new threads. It is unknown if this was the total amount of threads spawned, or if this simply was the limit of our console output. Other than this the system also gave us full access to Python and its features, but blocked us on the operating system level. Just like ideone, it blocked us from writing to the filesystem.

C.7.5. Relevance

We have concluded that none of the systems which currently exist are a suitable solution to our problem. Jupyter is unstable, allowing for ForkBombs which could also be blocked. Trinket does not run the actual Python interpeter, and as a result does not support modules which we need to support. Both ideone and Try It Online are well implemented solutions to the problem of creating an online IDE. Their implementation has enforced our believe that we should support the full Python interpreter, and not use something such as PyPy. However, our system is not simply an online IDE, and has extra requirements such as support for hidden tests and writing to file.

C.8. Safe Testing

In WebLab, user written code is tested against either user written tests or against so-called specification tests. The specification tests are hidden from the user, and any output of them is suppressed. The user only gets to see the amount of tests they passed and the amount of tests there are in total. These specification tests are used by WebLab to determine the grading when the code is reviewed automatically.

If a user is able to determine the content of the specification tests, they gain an advantage over their peers. Knowing what your program will be tested on and what it won't be tested on, can help the user in writing their code, and possibly even allows them to tamper with the tests itself. While manual review would be quick to find this tampering, a reliable and autonomous testing system is a requirement of our system. This brings us to the research questions "How can we test the users code without these tests becoming known to the user?" and "How can we test the users code without them being able to tamper with the test results?".

C.8.1. Keeping tests secret

The main measure that is used to keep specification tests secret, is by not giving the user any output of the program. The only thing that the user sees is the amount of passed tests and the total amount of tests. We would only have to ensure that no other information besides those two numbers can be displayed to the user. It doesn't even matter if the user manages to break out of the sandbox. As long as the user does not have access to the main server, there is no way to get hold of the specification test code itself. When the user is not running specification tests and is able to see program output, the code for specification tests will not be available on the virtual machine. This further prevents the user for attaining the code for the tests itself.

C.8.2. No tampering

Preventing users from being able to tamper with test results is quite a different story. As we have already outlined in section C.3, sandboxing Python in the language itself is unfeasible. Section C.2 also shows that there are many ways around any kind of sandboxing. Unfortunately, there are very few sources available on this topic.

This ultimately means that we cannot trust the Python runtime to determine the end results. We have explored different possibilities to separate the test code from the user code but these each have different problems. One example is to tell the user program to run on specific input, and to return their output. We then check if that output is correct with a different program. Since the Python runtime and the code that validates the output are separated from each other, the user would be unable to use any Python exploit to fake their test results. The problem with this method is that it requires the user code to output something that can be properly described as a string. Any object would need a proper toString method that can be used to test equality of objects. It would also limit what test code could test and what it cannot test. E.g. things like testing if certain methods get executed would be impossible.

A much better solution would be when the Python runtime doesn't even know what tests should pass and what tests should fail. This can be achieved by adding a number of tests that should always fail and a number of tests that should always pass. If the user code passes a test that should be impossible to pass, we immediately know that the user is trying to tamper with the test results. This concept is similar to the concept of zero-knowledge proofs [18].

The tests that we can add range from very simple checks to more elaborate ones. Examples of very simple

tests are `assertTrue(True)` and `assertFalse(True)`. More elaborate tests could also include calls to the user code to make them less distinguishable from other tests. For optimal security, at least some of these tests should be made to be almost indistinguishable from the real specification tests. An example would be an "inverse" test. If there is a specification test that has `assertTrue(x)`, we can create an identical test that has `assertFalse(x)`. For the Python program, these tests will be almost indistinguishable. As it is not possible for both of these tests to pass it allows us to detect any tampering.

The larger the number of additional tests, the harder it becomes to guess the correct solution. However, the larger the amount of (complex) tests the longer the required execution time would be to evaluate user code. We thus need to pick a good number of additional tests that makes it unlikely enough to guess the correct test outcomes. In the worst case, where the user is able to distinguish between (simple) fake tests and real tests reliably, only the "inverse" tests would still be a problem to the malicious user. If we cannot generate enough tests automatically, we might need the course coordinator to add specific tests that should fail for correct implementation.

It is important to note that we need only one wrong guess to determine that the user is tampering with test results. We could add a flag to the user's solution when we have detected tampering on earlier runs. Another possible sign would be if the user passes a different set of specification tests without altering their code. This could, however, also happen when the user uses any kind of randomisation.

By combining the methods outlined above, we can create a system that is capable of detecting tampering and that can decrease the likelihood that a user guesses the correct outcome for the specification tests. By flagging users that have been identified as tampering with results, we can make this system more secure by letting a reviewer check if the results are obtained legitimately.

C.9. Design

In this section we will provide an initial design for our system based on our research. This design consists of multiple protection layers, an instance manager and a concept for the test execution. These aspects are described in subsection C.9.1, subsection C.9.2 and subsection C.9.3 respectively.

C.9.1. Protection Layers

A single protection layer provides low protection to vulnerabilities as a single exploit can be used to breach the entire system. Therefore we will use a layered approach to ensure maximum security. A graphical overview of this layered system can be seen in Figure C.1.

- The inner layer is the layer in which the user code is ran and is therefore build around a compiler or interpreter. This layer can make use of hardened compilers and language specific safety features to make this execution itself more secure. For Python these measures, described in section C.3 and section C.2, consist of a non-limited version of the Python shell to ensure maximum compatibility and usability. Based on course settings, certain Python modules will be made unavailable by removing them from the library on this layer. We might also offer adapted Python modules to disallow people from using certain unsafe functions, while still offering needed methods from said modules.

- To sandbox this inner layer, Docker containers will be used. Each user will get its own container to separate the execution of his code from that of other users. For this container multiple settings, described in subsection C.1.1, will be used to limit the execution time, memory usage, available system calls and accessible file system.

- A third layer is used to make the operating system, on which the containers are created, more secure. This layer consists of kernel hardening tools that can be found in paragraph C.1.1. This way, a user that escaped its container can do less harm to the server and other containers (and thus users).

- To protect the server even more from such an escaped user, a virtual machines will be used to host the hardened operating systems from the previous layer. As can be seen in subsection C.1.2, it is not feasible to run a VM for every user, therefore this machine hosts multiple container instances.

- The outer layer is the server itself and hosts a pool of those virtual machines. This pool can also run on multiple physical servers to scale the system capacity.

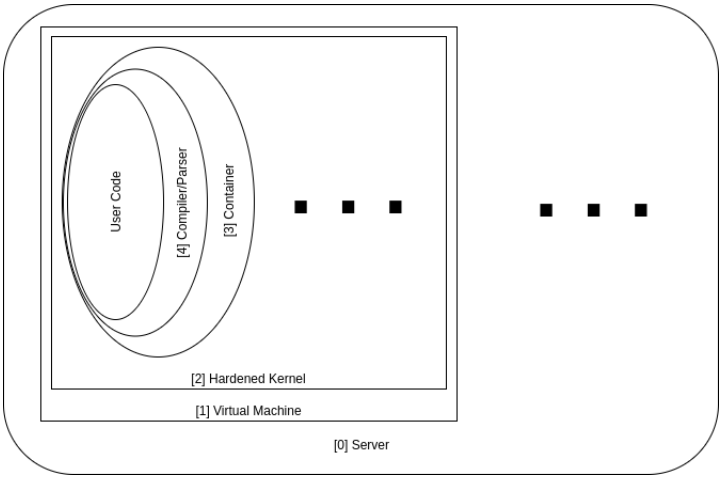


Figure C.1: A graphical overview of the protection layers in our system

C.9.2. Instance Manager

As we will use many container instances running on multiple VMs, we need a way to manage and distribute the instances. First, we need to use the correct number of virtual machines as described in subsection C.6.4. We then need to distribute the users and their containers over those VMs.

C.9.3. Testing

Whenever user tests are ran, the user test code will be executed directly, and program output will be sent back to the user. Example user test code is shown in Listing 6.

```
import unittest
import solution

class TestSolution(unittest.TestCase):
    def testName(self):
        sol = solution()
        sol.someMethod(someArgument)
        self.assertTrue(sol.someOtherMethod())

    # ... other tests ...

unittest.main()
```

Listing 6: Sample code for user tests

When specification tests are ran, we will use a completely different system. Outside the Virtual Machine, the specification tests will first be prepared. Additional tests will be added to verify that the user is not trying to tamper with the results, as described in subsection C.8.2. Tests are then shuffled and method names are changed to 'test1', 'test2' etcetera. This makes sure that a program created to pass one test run does not necessarily pass the next run. We will keep track of what tests are actual specification tests, and what tests are extra tests. We then put the created Python test file inside a user container, and start the Python program to run the tests. This Python process will output the test number and if that test passed or failed, as can be seen in Listing 7.

```
1 failed
2 passed
3 passed
```

Listing 7: Example output of Python process running specification tests

We then process these results (outside the VM), to determine the test score as well as if any tampering occurred. The test score is then sent to the user. For course assistants, we could also output the names of the tests that failed. If desired, we could also send the actual output of the Python code as detailed information.

D

Infosheet

Title of the project	Safe Execution of Native Code in WebLab
Name of the client organization	WebLab
Date of the final presentation	July 4, 2017
Final report	`http://repository.tudelft.nl`

Product description: WebLab, an online tool that provides a way to create and automatically test code, currently only supports JVM languages. The goal of the project was to extend the system so that other languages could be supported. The primary focus was to add support for Python, but flexibility of the system was required for future expansion.

Challenge: As WebLab is also used for exams, our system needed to support many users at the same time while retaining stability, safety and performance.

Research: In order to meet the requirements for the system, we needed to perform research on the following topics: Linux process containment, Python exploits and safe testing.

Process: During the majority of the development process we used Scrum to ensure flexibility and a focus on a working system. In these weeks we performed daily sprints, without maintaining the overhead Scrum enforces for evaluating the process. No unexpected challenges arose, resulting in a smooth development process.

Testing: We performed many load tests to test the scalability and performance of the system. We also tested various exploits and other malicious code to test the system security and reliability.

Outlook: The final product allowed for the execution of many different languages on WebLab, but we only created direct support for the Python programming language. For this language, our system can directly be deployed and used by the client. We provided a guide for adding other languages and a set of recommendations for possible extensions of the system like image based output.

Team

Taico Aerts

Role	Developer, Scrum master
Interests	Parallel Computing, Concurrency, Performance optimisation

Chiel Bruin

Role	Developer
Interests	Low level Linux, Programming languages

Bram Crielaard

Role	Developer, Product owner
Interests	Software Security, Exploit Discovery, Operating Systems

All team members contributed to the reports and the final presentation.

Client, Coach and Contact

TU coach	Prof.dr. E. Visser	*Programming Languages Group, TU Delft*
Client	Ir. D. M. Groenewegen	*WebLab*
Contacts	Taico Aerts	devtaeir@taico.nl
	Chiel Bruin	chieljkbruin@xs4all.nl
	Bram Crielaard	bram@crielaard.co.uk

E

Original Project Description

This appendix includes the original project description as listed on BEPsys. This description was what made us choose this project, with the title *Safe native code execution in WebLab back-end*, at the beginning of the quarter.

E.1. Project Description

The WebLab programming education environment allows students to write and execute solutions to programming assignments in a web interface. Code is executed and tested on the server. It is crucial that this execution is safe, i.e. does not affect the integrity and availability of the server. In order to achieve this, currently code is executed in the Java Virtual Machine, which allows restricting the capabilities of code. Unfortunately, this limits WebLab to programming languages that can be executed on the JVM; currently WebLab supports Java, Scala, C (via a translation to JavaScript), JPython (python for the JVM), and JavaScript. We would like to extend this set beyond languages that run on the JVM. In particular, Python is adopted in many courses at TU Delft and there is a demand for WebLab support for such courses. However, the typical applications of Python in data analytics requires libraries such as numpy, which rely on libraries written in C.

The goal of this project is to investigate and develop a back-end for WebLab that supports non-JVM languages, yet is safe.

E.2. Company Description

The TU Delft PL group conducts research into concepts and techniques for programming language design and implementation. The Academic Workflow Engineering team of the PL group applies advanced programming languages techniques to the development of software to automate academic processes.

E.3. Auxiliary Information

Company	TU Delft
TU Delft coach	Danny Groenewegen
TU Delft coach email address	d.m.groenewegen@tudelft.nl
Company contact	Eelco Visser
Company contact email	e.visser@tudelft.nl

F

User Manual

This chapter lists all the information that is needed for a system administrator to maintain the system. First section F.1 describes the process of creating and optimising new images, deploying them to the running machines, and in case of a new language constructing a testing framework for it. In section F.2 the methods are shown to properly set up and configure the machines. Then section F.3 shows the possibilities of the system settings to change the behaviour of the backend. Lastly a list of pointers will be given to documentation of functionalities described in this chapter for quick reference.

F.1. Building Images

As the backend is based on running tasks in specialised Docker containers, building and configuring of those containers is a powerful tool to expand the functionalities of the system. These updated container can be used for supporting more languages than the Python3 that is currently supported. But in order to fully integrate these new images in the backend, some components must be created. The methods of doing this and the interfaces they must follow, are described in the following sections.

F.1.1. Build

Images are built from a file describing the build process of said image, called a `Dockerfile`. In this file, a base image is given on which the image is expanded. As we already stated, the final image is preferably as small as possible. Therefore it is advisory to use as small as a base image as possible. One of the tinycore Linux bases is, when available (see Docker Hub), the best starting point. A second measure to decrease the size is to remove all the files required for the setup (for example a package manager) after installing the setup. When the installation of all the needed files is done, the user's home folder should be constructed at `/user_code/`. This folder must also contain an output folder `/user_code/output` in where the user can create the output files. This output folder should also be the working directory of the user. Lastly a command should be added that must run when the container starts. With this a build file is constructed that looks something like the one shown in Figure F.1.

```
# Use an official Python runtime as a base image
FROM tatsushid/tinycore-python:3.6

# Set the working directory to /user_code
WORKDIR /user_code

# Copy any files that must be packaged
ADD . /user_code

# Install any needed packages
RUN easy_install pip;
RUN pip install --no-cache-dir -r requirements.txt

# Remove all unneeded files
RUN sudo pip uninstall pip -y
RUN sh ./remover.sh
RUN rm Dockerfile
RUN rm requirements.txt
RUN rm rmmodules.txt
RUN rm remover.sh

# Set up permissions for user folder
RUN adduser --system -s /sbin/nologin student
RUN chown student: /user_code
RUN chown student: /user_code/weblabTestRunner.py
RUN chown student: /user_code/solution.py
RUN chown student: /user_code/test.py
RUN chmod u+w /user_code
USER student

# Define environment variable
ENV NAME python
ENV HOME /user_code

RUN mkdir /user_code/output

WORKDIR /user_code/output

# Run test.py when the container launches
CMD python ../test.py > stdout.txt 2> stderr.txt
```

Figure F.1: An example of a Dockerfile

When the file is created, the image can be build on a machine so that this machine can use the image. Building such images can be done using the Docker build text command. This command follows the following pattern:

```
~/ $ docker build -no-cache -t <LANGUAGE>:<VERSION> <DIR>
```

F.1.2. Deploy

Each of the images used must be present on the machines that run the user tasks (see subsection 3.10.2). These images can be deployed on these servers in three ways. The first two are based on a repository that contains all the images ready for use, while the third is based on building fresh installs of them on the machine. First of all we must configure the Docker daemon to be able to directly communicate with the machine. This is done by setting the Docker system variables to point at said machine. Luckily, docker-machine provides

a simple tool to perform this setup step. Assuming a machine is running with the name `exampleMachine`, the command

```
~/ $ eval $(docker-machine env exampleMachine)
```

is used to setup the Docker daemon to work with that machine. Now a pre-built image can be pulled to that machine using the Docker pull text command. By default this pulls from the Docker Hub registry. The command also supports pulling from a local registry that contains custom images. As creating such a registry creates some overhead, but pulling becomes quicker, this solution is best fitted when working with multiple machines. For a single machine, it will be easier to directly build the machines to the machine instead of building it on the registry.

F.1.3. Customise Images

The image currently used is `tinycore-python:3.6`, based on Python 3.6 and tinycore Linux. This installation is extended by `numpy` and `matplotlib` to create a small but versatile image that is sufficient in most cases for running Python. When needed this image can easily be expanded or shrunk by adding or removing libraries. For this, the image builder makes use of a `rmmodules.txt` file for the modules to remove and a `requirements.txt` file for the modules to install. These files include a line for every module to be removed or added respectively, as is common practice in many Python projects. When These two files are constructed, the new image can be build using the commands listed above.

F.1.4. Test Fuzzing

Every programming language follows a different syntax. This means that our test-fuzzing code must be adapted to work with a new language. In order to achieve this, two components must be added by implementing two abstract classes. When this is done, the abstraction of the abstract classes takes care of fuzzing the tests and parsing the results. The first class is an extension of the `Parser` class. This class takes a file containing the source code of the tests and splits this in various parts. These parts are the pre-test, test and post-test. The pre- and post-test are the parts of the code that should not be fuzzed and prepended or appended around the fuzzed tests. As no operation is required on these parts of code, they are stored as simple `String` objects. The test section are the to be fuzzed test cases. To enable the fuzzer to correctly generate the new test cases, this section should be parsed to a list of `TestMethod` objects. These objects contain the body of a method, the method signature and the signature of the end of the method. With the source parsed to this format, the actual fuzzing can be performed. This is done by the second class that needs to be implemented. This extension of the `Fuzzer` class must be able to invert a test case when given one. When for some reason a given test file cannot be inverted, for example when testing for exceptions, `null` should be returned. With these two classes implemented, the only thing left is to add these two new classes to the settings of the language. Without these settings the backend does not know which parser and fuzzer to use for the newly added language and as an effect cannot apply the test fuzzing.

F.1.5. Result Gathering

The output of a container is based on the files that are present in the `<working_dir>/output` folder. This folder should at least contain three files. The first two files, `stderr.txt` and `stdout.txt`, contain the console output of running the task. A simple way to achieve this is to pipe this output directly from the `CMD` field in the `Dockerfile` by appending `> stdout.txt 2> stderr.txt`. The third file must contain the results of the tests. Currently only [redacted] is supported for this file, as the parser in the backend is build for this format. The parsing can however simply be adapted to another format by creating a new result parser. The [redacted] follows a simple format to display all the test results as shown in F.2. Each `testResult` element has the attribute `name` in where the name of the test case is stored and contains a `success` field describing the test result. This field can contain any of the values from `OK, ERROR, FAIL, SKIP, EXP_FAIL, UNEXP_SUCCESS`. In addition to this result field there can be a `reason` field with some additional information about the test. This tag is required with the `success` values `FAIL` and `ERROR` and is omitted in the other cases. It must also be noted that the `ERROR` value should provide a reason, but as this result was caused by an error rather than a failure, it should be written to `stderr` instead.

Figure F.2: The layout of the result file

As this format is required for retrieving the results from a container, any new images that do not use the `weblabTestRunner` provided by our backend should implement a way to create these files from the test runner. In the case of Python, the `unittest` framework provides a way to attach a custom Python testrunners that runs all the test cases. This was used to replace the original runner with one that writes the results to [redacted] rather than to the standard output. As long as the newly constructed test runners for a new language create a valid [redacted] that follows the syntax described above, the test results will automatically be parsed when the results of a container are received.

F.2. Machine Configuration

The backend supports running the tasks over multiple machines. For this to work, some configuration is needed on the machine side. Most importantly a Docker Docker daemon must be running on the machine. Without this running docker-machine cannot connect to it and thus we cannot execute any tasks on that machine. To make the setup of Docker on the machine easier, docker-machine provides docker-machine drivers for many types of (virtual) machines including Amazon Web Services, Microsoft Azure, Google Compute Engine, Oracle Virtualbox and a generic driver that should work on most other types of servers. These drivers will check if Docker is present and if not, install it. When the machine is set up to accept tasks, the machine must be added to the backend settings for the backend to start scheduling tasks on it. How these settings can be applied is described in section F.3.

F.2.1. Kernel Patching

To make the machine more secure the operating system could be hardened. This way a user that was able to, somehow, escape its container, has less chance of causing harm to the machine by the use of (kernel) exploits. As can be seen in section C.1.1, the best way to perform these hardening steps is by applying the grsecurity kernel patches. In the case of RancherOS, we created a script that performs this patching of the operating system automatically. Running this script can be done by executing

```
~/ $ sudo ./RancherOS/OS\builder/build.sh <PATCH_FILE>
```

in the root folder of the repository. In the folder `/RancherOS/OS\builder/pathces` , a patch file present containing the configuration of the grsecurity settings. For any other operating system the process should be similar.

F.3. System Configuration

To configure the backend to the needs of the moment, a settings file is present in the working directory. Saving this file instantly updates the settings of the backend. Please look at the log files after applying new settings, as the new values will be validated and omitted when marked as invalid. When deploying a new machine, adding the settings of said machine is enough for the backend to start scheduling tasks on it. When the backend is run, a file called `settings_example.yml` is placed in the working directory, alongside the settings file, containing all the possible settings, their description and default values.

F.4. Command-line Interface

Besides the setting as an interface to the backend, there is also the possibility to access system functionality using the command-line interface (CLI). As the client preferred to use a settings file, the development of the CLI was mostly aimed at providing quick debug tools like changing the log levels and displaying system status. Due to the modular setup of the CLI it is easy to add new functionalities by implementing a `Command` class and attaching it to the CLI. The most useful commands are the `help` command to show all available commands and command usage, `status` for displaying the system status, `log` to change the log levels and `exit` to safely shutdown the backend.

F.5. Pointers

Docker build text

`https://docs.docker.com/engine/reference/commandline/build/`

Docker pull text

`https://docs.docker.com/engine/reference/commandline/pull/`

Docker API

The API used to communicate with the containers used by the Spotify docker-client API. `https://docs.docker.com/engine/api/v1.29/`

Docker Hub

`https://hub.docker.com/`

Docker run

`https://docs.docker.com/engine/reference/run`

Docker-machine drivers

`https://docs.docker.com/machine/drivers/`

Dockerfile

`https://docs.docker.com/engine/reference/builder/`

Grsecurity

`https://www.grsecurity.net/`

Python testrunners

`http://python.net/crew/tbryan/UnitTestTalk/slide30.html`

RancherOS customisation

Compile the kernel: `https://docs.rancher.com/os/custom-builds/custom-kernels/`
Compile the OS: `https://docs.rancher.com/os/custom-builds/custom-rancheros-iso/`

Spotify docker-client API

`https://github.com/spotify/docker-client`

Bibliography

[1] Ned Batchelder. Eval really is dangerous, June 2012. URL https://nedbatchelder.com/blog/201206/eval_really_is_dangerous.html. [Online; accessed 8-May-2017].

[2] Ned Batchelder. Finding python 3 builtins, February 2013. URL https://nedbatchelder.com/blog/201302/finding_python_3_builtins.html. [Online; accessed 8-May-2017].

[3] Padraig Brady. timeout(1) - linux manual page, 2017. URL http://man7.org/linux/man-pages/man1/timeout.1.html. [Online; accessed 27-May-2017].

[4] Hao Chen and David Wagner. Mops: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244. ACM, 2002.

[5] Mike Cohn. *Succeeding with Agile: Software Development Using Scrum*. Pearson Education, 2009.

[6] Philippe De Ryck, Maarten Decat, Lieven Desmet, Frank Piessens, and Wouter Joosen. Security of web mashups: a survey. In *Nordic Conference on Secure IT Systems*, pages 223–238. Springer, 2010.

[7] Docker. Docker machine overview - docker documentation, 2017. URL https://docs.docker.com/machine/overview/. [Online; accessed 11-May-2017].

[8] Docker. Docker overview - docker documentation, 2017. URL https://docs.docker.com/engine/docker-overview/#the-underlying-technology. [Online; accessed 11-May-2017].

[9] Docker Inc. Use the docker command line, 2017. URL https://docs.docker.com/engine/reference/commandline/cli/. [Online; accessed 10-May-2017].

[10] Docker Inc. docker inspect - docker documentation, 2017. URL https://docs.docker.com/engine/reference/commandline/inspect/. [Online; accessed 11-May-2017].

[11] Docker Inc. What is docker?, 2017. URL https://www.docker.com/what-docker. [Online; accessed 15-June-2017].

[12] Docker Inc. What is a container?, 2017. URL https://www.docker.com/what-container. [Online; accessed 15-June-2017].

[13] Docker Inc. docker-machine inspect - docker documentation, 2017. URL https://docs.docker.com/machine/reference/inspect/. [Online; accessed 11-May-2017].

[14] Jake Edge. The failure of pysandbox, November 2013. URL https://lwn.net/Articles/574215/. [Online; accessed 8-May-2017].

[15] Python Software Foundation. 26.4. unittest — unit testing framework, June 2017. URL https://docs.python.org/3/library/unittest.html. [Online; accessed 21-June-2017].

[16] The Python Software Foundation. 2. built-in functions, April 2017. URL https://docs.python.org/3/library/functions.html#eval. [Online; accessed 9-May-2017].

[17] Michael Fox, John Giordano, Lori Stotler, and Arun Thomas. Selinux and grsecurity: A case study comparing linux security kernel enhancements. , 2009.

[18] S Goldwasser, S Micali, and C Rackoff. The knowledge complexity of interactive proof-systems. In *Proceedings of the Seventeenth Annual ACM Symposium on Theory of Computing*, STOC '85, pages 291–304, New York, NY, USA, 1985. ACM. ISBN 0-89791-151-2. doi: 10.1145/22145.22178. URL http://doi.acm.org/10.1145/22145.22178.

[19] Ilya Grigorik. Image optimization, February 2017. URL https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/image-optimization. [Online; accessed 12-May-2017].

[20] Danny M. Groenewegen, Zef Hemel, Lennart C.L. Kats, and Eelco Visser. Webdsl: A domain-specific language for dynamic web applications. In *Companion to the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, OOPSLA Companion '08, pages 779–780, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-220-7. doi: 10.1145/1449814.1449858. URL http://doi.acm.org/10.1145/1449814.1449858.

[21] TU Delft PL Group. Bepsys project description, 2017. URL https://bepsys.herokuapp.com/projects/view/271. [Online; accessed 4-May-2017].

[22] GRSeurity. grsecurity - features, 2017. URL https://www.cyberciti.biz/faq/understanding-bash-fork-bomb/. [Online; accessed 11-May-2017].

[23] Jesse Hertz. Abusing privileged and unprivileged linux containers, 2016. URL https://www.nccgroup.trust/uk/our-research/abusing-privileged-and-unprivileged-linux-containers/. [Online; accessed 11-May-2017].

[24] Kaazing. websocket.org - powered by kaazing, 2011. URL http://www.websocket.org. [Online; accessed 20-June-2017].

[25] Maxwell N Krohn, Petros Efstathopoulos, Cliff Frey, M Frans Kaashoek, Eddie Kohler, David Mazieres, Robert Morris, Michelle Osborne, Steve VanDeBogart, and David Ziegler. Make least privilege a right (not a privilege). In *HotOS*, 2005.

[26] Agile Business Consortium Limited. Moscow prioritisation, 2014. URL https://www.agilebusiness.org/content/moscow-prioritisation. [Online; accessed 28-April-2017].

[27] Linux kernel documentation. Cfs bandwidth control, 2017. URL https://www.kernel.org/doc/Documentation/scheduler/sched-bwc.txt. [Online; accessed 27-May-2017].

[28] Linux man pages. man page ulimit section 1, 2009. URL http://www.manpagez.com/man/1/ulimit/. [Online; accessed 10-May-2017].

[29] Linux man pages. renice(1) - linux manual page, 2014. URL http://man7.org/linux/man-pages/man1/renice.1.html. [Online; accessed 10-May-2017].

[30] Linux man pages. cgroups(7) - linux manual page, 2016. URL http://man7.org/linux/man-pages/man7/cgroups.7.html. [Online; accessed 10-May-2017].

[31] Linux man pages. seccomp(2) - linux manual page, 2016. URL http://man7.org/linux/man-pages/man2/seccomp.2.html. [Online; accessed 11-May-2017].

[32] David MacKenzie. nice(1) - linux manual page, 2016. URL http://man7.org/linux/man-pages/man1/nice.1.html. [Online; accessed 10-May-2017].

[33] Ming Mao and Marty Humphrey. A performance study on the vm startup time in the cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 423–430. IEEE, 2012.

[34] nixCraft. Understanding bash fork() bomb, 2007. URL https://www.cyberciti.biz/faq/understanding-bash-fork-bomb/. [Online; accessed 10-May-2017].

[35] Jérôme Petazzoni. Lxc, docker, security: is it safe to run applications in linux containers? LinuxCon 2014, 2014. URL https://www.slideshare.net/jpetazzo/is-it-safe-to-run-applications-in-linux-containers.

[36] NSA Peter Loscocco. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference. Boston: USENIX Association*, 2001.

[37] Gabe Pike. Escaping a python sandbox with a memory corruption bug, March 2017. URL https://hackernoon.com/python-sandbox-escape-via-a-memory-corruption-bug-19dde4d5fea5. [Online; accessed 8-May-2017].

[38] The PyPy Project. Frequently asked questions, 2016. URL https://pypy.readthedocs.io/en/latest/faq.html#module-xyz-does-not-work-in-the-sandboxed-pypy. [Online; accessed 8-May-2017].

[39] The PyPy Project. Pypy's sandboxing features, 2016. URL http://doc.pypy.org/en/latest/sandbox.html. [Online; accessed 8-May-2017].

[40] Linda Rising and Norman S Janoff. The scrum software development process for small teams. *IEEE software*, 17(4):26–32, 2000.

[41] Ronald L Rivest et al. Chaffing and winnowing: Confidentiality without encryption. *CryptoBytes (RSA laboratories)*, 4(1):12–17, 1998.

[42] Saumil Shah. Hacking with pictures, 2014. URL https://de.slideshare.net/saumilshah/hacking-with-pictures-hacklu-2014. [Online; accessed 20-June-2017].

[43] William Stallings. *Operating Systems: Internals and Design Principles| Edition: 8*. Pearson, 2014.

[44] Victor Stinner. [python-dev] the pysandbox project is broken, November 2013. URL https://mail.python.org/pipermail/python-dev/2013-November/130132.html. [Online; accessed 8-May-2017].

[45] Tatu Ylonen Timo Rinne. scp(1): secure copy - linux man page, 2014. URL https://linux.die.net/man/1/scp. [Online; accessed 11-May-2017].

[46] Filippo Valsorda. Escaping a chroot jail/1 | pytux, 2013. URL https://filippo.io/escaping-a-chroot-jail-slash-1/. [Online; accessed 12-May-2017].

[47] VMware Inc. Virtual machine communication interface (vmci) api reference documentation, 2007. URL https://pubs.vmware.com/vmci-sdk/. [Online; accessed 11-May-2017].

[48] Chris Lalancette Laine Stump Daniel Veillard Dani Coulson David Jorm Scott Radvan W. David Ashley, Daniel Berrange. Libvirt application development guide using python, 2017. URL https://libvirt.org/docs/libvirt-appdev-guide-python/en-US/html/. [Online; accessed 11-May-2017].