Investigating Application Compatibility in Prototype Operating Systems

Vlad-Radu Schiller

Supervisor: Dr. Pierre Olivier

University of Manchester

March 2023

Abstract

Ensuring application compatibility is a critical issue that needs to be addressed by any operating system. Despite the availability of several solutions to tackle this problem, most of them are not effective when it comes to new operating systems, such as unikernels. To address this issue, this project aims to investigate the problem of application compatibility. To achieve this goal, I contributed to Loupe, a tool that helps create an optimal support plan for operating system developers regarding the system calls used. My work has been integrated into a larger research effort, and has been submitted to an A*-ranked conference. Afterwards, I have applied my acquired knowledge in the context of Unikraft by creating a comprehensive test suite for all Linux system calls, inspired by the Linux Test Project (LTP). I created an automation script for all of the tests from LTP, as well as a central module, responsible for running the tests and displaying their results. The test suite aims to ensure that Unikraft's implementation of Linux system calls is compatible with the applications that depend on them. This work could help ensure that Unikraft - as well as other prototype operating systems - can support a wide range of applications, making them more viable and popular operating systems.

Acknowledgements and Thanks

I would like to thank my supervisor, Dr. Pierre Olivier, who was incredibly helpful throughout the duration of the project, especially in the beginning, when he helped me understand how to use Loupe and Unikraft. Afterwards he gave me a lot of great suggestions whenever I would get stuck, and he generally ensured that my work is worthy of a third year project for a prestigious university such as this one. He was also very supportive and appreciative of my new ideas for the project, thus motivating me to be ambitious and to think outside of the box. I would also like to thank the Unikraft community for being so supportive and helpful, when I needed it most.

Table of Contents

1 Introduction

1.1 Context

An operating system is only as good as the applications it supports. In this regard, already established operating systems such as Windows, MacOS and Linux have a great advantage over novel operating systems. In order to have the chance of gaining traction, new operating systems have to implement a series of compatibility options in order to support as many applications as possible. This project aims to investigate these solutions and contribute in streamlining the process of dealing with application compatibility. In this regard, I had the opportunity of contributing to a larger research effort about Loupe, a highly regarded and novel tool. This research has even been submitted at top tier (A*) conferences such as ASPLOS. This tool, among others, is able to take an application list and generate an optimal support plan for the developers to most efficiently invest their efforts. I also used my contributions to help the Unikraft operating system, which reinvents the way an operating system works by creating a separate image for every application it needs to run. This makes Unikraft an extremely versatile and modular system, claiming boot times of just a few milliseconds and memory usage of just a few megabytes of RAM. Unikraft and other operating systems like it (known as unikernels), have great potential, especially in the cloud computing industry, and this project aims to help them, as well as other prototype operating systems, achieve their potential.

1.2 Motivations and Aims

Starting with my second year of university, I switched my main operating system from Windows to Linux. This has kickstarted a lasting interest towards virtualisation technologies and operating systems in general. This project represents a great opportunity for me to learn. The original title of the project was about “contributing to Unikraft in a way that is worthy of a third year project”. However, after expressing my interest about compatibility, I decided alongside my supervisor that this work can go further than just Unikraft, to any prototype operating system. Since the project’s list of aims and objectives was vague and not clearly defined, I had to set a few aims myself (in accordance with my supervisor), as well as a way

to quantitatively measure my success. At the same time, since my project will be split into two parts, so will the objectives. These are shown below:

Objective	Part 1	Part 2
Learn about the environment: This is a crucial step in the project. By learning and getting accustomed with Loupe, I will be able to contribute to Loupe directly, as well as gain important insight that will help me in the next parts.	I will consider this completed if I manage to learn and understand how to use Loupe.	I will consider this step completed if I can properly utilise Unikraft, and understand the Linux Test Project.
Apply the newly gained knowledge in a specific scenario: The previous step was more theoretical in nature. This step will provide a lot more practical and quantifiable results than the last step, and my success in one will also serve as proof of my success in the other.	I will consider this point achieved if I manage to directly help the Loupe developers in a meaningful way.	I will consider this point achieved if I manage to use my knowledge in order to create something that can help the Unikraft developers and prototype operating system developers in general.
Visualise the results: Working with a lot of data can make it easy to misinterpret and hard to manage, especially if the project is intended to be used by a team of developers in the future. This step will make it easier for the Unikraft and Loupe developers to work with my results.	I will consider this point completed if I am able to turn the resulting data from my experiments into something visual and tangible.	

Due to their chronological structure, with one step depending on the previous, these aims should thoroughly cover all of the main points needed to achieve good results and to understand where to improve, should the objectives not be achieved.

2 Background and Theory

2.1 What is a Unikernel?

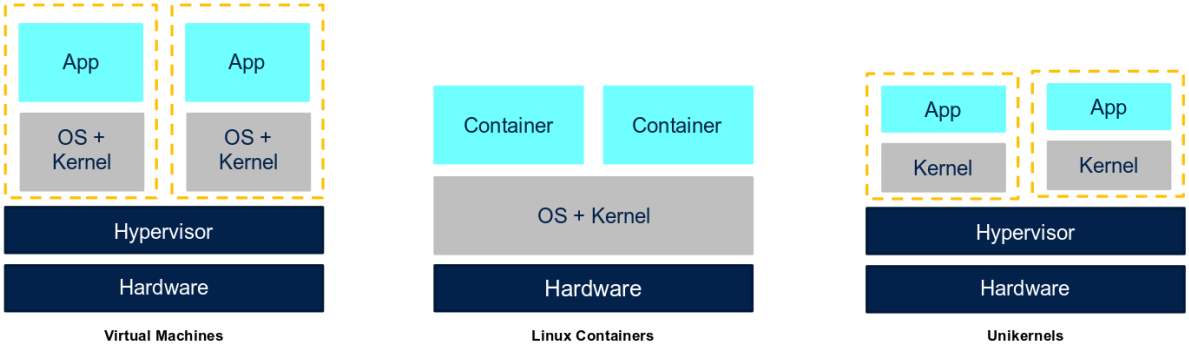


Figure 1: The design of unikernels, virtual machines and containers

When it comes to virtualisation solutions, two classic, tried and true options stand out from the rest. The first one is a virtual machine, which emulates a certain operating system, with the user being able to use almost all of the operating system's features. This solution virtualizes an entire hardware stack alongside the host operating system, which can make it lose a lot of performance. However, the more advanced configurations that use a virtual machine use what is known as a hypervisor, an extremely lightweight operating system that makes the performance loss almost minimal [1]. For Linux, the most common solution is KVM [2] and QEMU [3], because they offer the best of both worlds by turning the host operating system into a hypervisor. By doing so, setups that rely on KVM can run multiple operating systems at the same time, using the Linux kernel as a hypervisor.

The second most commonly used option is represented by containers, such as the ones created with the help of Docker [4]. Docker works through the use of docker files, which act as a stand-alone independent package that contains everything needed for an application or a set of applications to work. Unlike virtual machines, containers share the host's kernel in order to function, but it is much easier to run a set of containers at the same time, because they are a lot more lightweight compared to virtual machines. More recently, a newer and better solution aims to implement the best of both solutions, and is known as a unikernel.

A unikernel is a specialised, single address space operating system [5]. It reimagines the way that an operating system works, by focusing on the app, as opposed to a certain environment. A unikernel builds an image that is specifically tailored for a single app, relying on a lightweight hypervisor such as KVM to run more than one program simultaneously. Figure 1 shows the virtualisation layers needed for a unikernel compared to a virtual machine and a Linux container.

One of the best unikernels at the present is Unikraft [6]. This open source project is a flexible and lightweight development kit for unikernel images. It is modular, customisable and easy to use. Because of this, Unikraft is one of the central parts of my project, and I will refer to Unikraft specifically when talking about unikernels.

2.2 Benefits and Drawbacks of Unikraft

As is the case with all ideas and implementations, unikernels have their advantages and disadvantages:

2.2.1 Advantages

Some users might be concerned about security or storage size, but by far the most pertinent aspect for the common user is how fast the operating system is. In this regard, unikernels are incredibly fast, easily surpassing virtual machines and containers. Unikraft in particular is the fastest unikernel when benchmarking for their set of applications [7].

At the same time, because of Unikraft's modular design, the resulting images can be very light, both in terms of processing power and in terms of storage space. On their official website, Unikraft claim a memory footprint of a few MB of RAM and boot times of less than 10 ms. This is incredibly important for cloud computing, as a more lightweight system is able to use the servers more efficiently, which reduces the cost for the server providers and the customers.

Unikraft's size is also a great advantage for security. Indeed, when counting for the fact that they don't rely on the host's kernel like Docker containers, unikernels are a very good

compromise between security and speed. Their smaller size also gives less access points for security breaches.

2.2.2 Drawbacks

Naturally, not even a system such as Unikraft is perfect. Most of Unikraft's drawbacks stem from the fact that this operating system is still at a very early stage of development, which can pose a number of different problems. For one, it is very difficult to enter the ecosystem of Unikraft, because there is not enough comprehensive documentation for every feature. At the same time, Unikraft is less tested compared to containers and virtual machines, which makes it more prone to bugs. On a similar note, the minimalist design of unikernels in general can make them more difficult to debug and manage, especially in production environments.

However, the most important drawback of Unikraft, which is also the main focus of this project, is the problem of application compatibility. This problem is, for instance, one of the main reasons why Linux will never become as popular as Windows. For Unikraft, this problem applies to an even greater degree, because of how new it is. This means that there are virtually no developers that make applications that are designed with unikernels in mind, even though, with their help, users can benefit from greater performance and security, as well as less resource usage, making it perfect for cloud computing. At the same time, the solutions to this problem need to let the target application work without any modification to the application itself, as this entails extra actions from the developer, and even if all that needs to be done is for the developer to recompile their app, this represents an extra hurdle that unikernels need to overcome.

2.3 Efforts Towards Prototype OS Application Compatibility

Application compatibility is a serious obstacle even for well established operating systems such as Linux. This is doubly true for unikernels, as well as prototype operating systems in general. Fortunately, there are a lot of solutions for this issue at the developer's disposal. For more established operating systems, there is the option of virtualisation, containerisation or even dual booting, allowing the user to switch to an operating system that is able to run the applications that are incompatible with the host operating system. This, however, does not

solve the core issue of application compatibility, and at the same time, the more an operating system relies on these methods for application compatibility, the more likely a user is to “cut out the middleman” and move to the guest operating system entirely. For prototype operating systems, there are other options that do not force the user or the application developer to invest any effort.

2.3.1 Emulation

The first option for developers is emulation. Through the use of emulators, an operating system can be made to behave like another, allowing applications that work for the latter to also work for the former. These work by virtualising the hardware and software stack in order to mimic a different system in which the target application functions. There are many different types of emulators, which focus on different components of the operating system. Emulators can also be both digital and physical, such as the Centris 610, a DOS-compatible hardware emulator for Macintosh computers [8]. In the case of Linux, a good example is QEMU [3], a staple of the virtualisation industry. Emulators are very popular for older gaming console systems too, with some operating systems only consisting of a suite of emulators, such as RetroArch [9] and EmulationStation [10].

2.3.2 Application Compatibility Layers

Another good solution is making use of application compatibility layers (or ACL). These software solutions consist of a translation layer, which intercepts the system calls made by the application and translates them for the operating system. Similarly to emulators, these solutions also mimic certain features of the target operating systems, such as libraries. This allows the target application to run in the ACL without any modifications. However, unlike emulators, ACL’s do not virtualise hardware. A popular implementation of this technology is the Linux Compatibility Layer made by the developers of FreeBSD [11]. This translation layer lets Linux applications work in the FreeBSD operating system. This approach is also used by the Unikraft developers to mitigate their application compatibility problem.

2.3.3 API Bridging

API bridging is a similar solution to an ACL, in that it features a compatibility layer that intercepts and translates the system calls of the target application and adapts them to the host

operating system. However, unlike an ACL, API bridging does not mimic other features specific to the target operating system, which makes API bridging a less comprehensive approach to this problem compared to an ACL, with a narrower range of uses. One of the most important and widely used API bridging tools is the Wine project, which offers a translation layer for Windows applications to work on Linux [12]. This project has generated a lot of forks, such as Valve's Proton compatibility layer [13], which makes almost their entire catalogue of games available to Linux users.

These solutions are usually standardised and implemented by a lot of dedicated developers. But what options do new developers have with an even lower amount of development resources? Is there a way to optimise their development efforts, especially in the early stages of development?

2.4 Loupe

Loupe is a tool in its early development towards which I dedicated a large part of my work for this project. It is currently used by the Unikraft developers to help with the porting of applications for their operating system. This tool has two main functionalities: porting applications and offering support plans. The first one is really useful for a developer that wants to analyse an application that is not already supported by their operating system, and the second feature is useful for the developers that have more than one app, by offering a bird's eye view of the optimal development procedure. Both of these features will be tested at length and improved throughout my project.

The first functionality, which will be presented in detail in the next chapter, takes an application and performs static and dynamic analyses on the application to ascertain its system call usage. At the end of this analysis, the tool outputs a list with all of the system calls that are used by the app. The output for the dynamic analysis includes which system calls can be stubbed and/or faked. These outputs are stored in a separate database folder, alongside all of the relevant files that were used in the test, in order for the porting process to be reproducible (shown in figure 2). applications can also be analysed with the help of a test suite, which offers a different set of results than the dynamic analysis.

This feature is not new, as there are tools such as strace that already perform it [14]. However, Loupe comes out on top because of their analysis method. Strace performs a static analysis, meaning that it analyses the application in an external environment without running it; while this solution works, this can lead to the tool vastly overestimating the required system calls. Loupe, alongside a static analysis, also performs a dynamic analysis, which means that it analyses the application as it runs. As opposed to the static analysis, this solution tends to underestimate the system calls that are needed, depending on how thoroughly is the target application tested. On the flip side, dynamic analyses can also account for which system calls can be stubbed and/or faked for the application to work. Since Loupe performs both types of analyses, it can offer a much more accurate and complete set of data for the user.

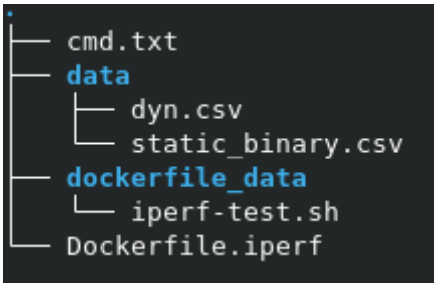


Figure 2: The sample file structure in the loupedb folder

The second functionality relies on the first one, and is extremely useful for developers. Using the information from the database, Loupe is able to generate the optimal order to implement, stub and fake system calls so as to have as many applications ported in the shortest time possible. This feature requires the user to input a set of applications that are already ported in the database, or can choose to work with all of the applications in the database. The tool can also offer a more visual output by plotting the results on a series of different charts.

2.5 Related Work

Although my project focuses more on the compatibility aspect of novel operating systems, unikernels and Unikraft more specifically played an important part in applying my work. Therefore, I read a couple of papers about this subject. Unikraft's paper [15] provides a lot of insight into the advantages and obstacles of their approach to unikernels. These two articles presenting MirageOS offer some more information about the rise and importance of

unikernels for cloud computing [5][16]. On a similar note, other unikernels such as HermiTux, OSv and MirageOS all offer alternative approaches to this operating system design. OSv will be referenced in my project [17], MirageOS is the first prototype unikernel [5], and HermiTux is the first unikernel that is designed around binary compatibility with Linux applications [18].

My main source of studying was on the topic of compatibility. This paper investigates and compares various compatibility solutions, including the options that were listed [19]. Loupe also has a prototype paper, but it is yet to be published as of writing this essay. This paper describes the functionalities of various virtualisation options, including virtual machines and containers [20].

In order to understand how Loupe works, I needed to understand how the Linux kernel works. Therefore, I read this paper, which presents the system call structure of the Linux kernel, as well as their functionalities [21]. This paper gives great insight into the difference between the static and dynamic analyses, which is an important point related to Loupe, even though it focuses on malware detection in the case of Android systems [22].

For the Linux Test Project, these papers [23][24], as well as their website [25], have given me a lot of insight into how the project works. In order to better understand how to use system calls inside C programs, I made use of this website [26], which provides a lot of information about the standard C libraries and other standard Linux applications.

This paper gives great insight into how the Unikraft ELF loader works, a critical component for the second part of my project [27].

3 Porting Applications with Loupe

3.1 System Call Analysis

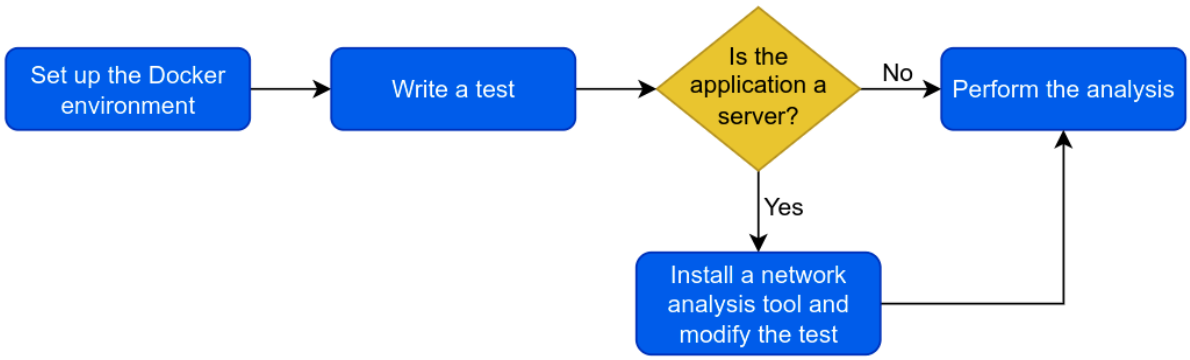
In the first semester, I chose to focus on an important tool that was used by the Unikraft developers: Loupe. As presented earlier, this tool is useful, having the potential to benefit a lot of operating system developers. In spite of this potential, it was still at an early stage in development, and thus did not have a significant amount of data for their listed benefits to be more than merely theoretical. I decided to dedicate my efforts towards changing this, by testing Loupe in another case.

3.1.1 Choosing the Application List

A lot of the applications that the Unikraft developers wanted were already ported. However, Loupe should work for as many new operating systems as possible. Because of this, myself and the Loupe developers decided to port the applications from the application list of the operating system OSv, another unikernel. Not only will this add a significant number of applications to the database, but this data can also be used to ascertain the usefulness of Loupe in a more categorical way. With the application list, we can more clearly see a difference in the development progress with and without the tool's help. In order to have a relevant sample size that was achievable, we set out to analyse 100 applications.

3.1.2 The Basics of Application Porting with Loupe

After setting the bar, it took me a few weeks to get used to the tool. After this period, I performed this procedure for each application (also detailed on their GitHub repository), shown in the following flowchart:



1. Make the application work in a Docker container. Because the tool analyses the target application in a Docker container, the first step is to make the application work in a stand-alone Docker container. From my experience, this step is usually the most difficult to achieve, as this step varies widely with every app, with it not being able to be standardised. Some applications proved to be easy to set up in a Docker container (such as a Java hello world app), while other, more complex applications may require a lot of effort to work properly. Some applications are outright impossible to install in a Docker container (Detailed further in section 3.3.)
2. Write a test to analyse the app. This step is very important, and it is also different on a case by case basis. When examining an app, Loupe runs the application many times and also runs a test, to determine if the application worked or not. Thus, the test needs to detect any errors that appear when executing the app, or expect a certain output from the app. From here, the procedure is slightly different if the application is a server application (such as MySQL) or a run to completion application (such as ffmpeg). Because these applications are different on a fundamental basis, they require a slightly different method of analysis.
3. If the application is a server app, install a network analysis application in the Docker container, such as “wrk” or “iperf”. The test written in the previous step needs to be changed slightly, looking at the expected output of “wrk” instead of the target app.
4. Determine the relevant binary to run and start the analysis. This can be a binary of the application itself (usually named main, or run, or the application name), or a secondary binary (such as python3, if the application needs it to run). If the application is a run to completion app, an extra parameter needs to be included in the explore.py execution. From here, if the results are satisfactory and there are no errors, Loupe should create a new folder with the analysis results, as well as the auxiliary files used in the porting process.

If step 4 is completed, then the application is ported. All that needs to be done now is to make a new commit for the “loupedb” folder. Loupe won’t normally add new entries in the database if the folder has modifications compared to the latest git commit, but this can be bypassed with a parameter.

3.1.3 Porting in Practice

The previous section presented an overview of the porting procedure. This section will present the more specific quirks that I encountered with specific examples, as well as mention the results of this experiment.

At first, the procedure of porting an application was very difficult for me, but it got easier as I understood more and more about how Loupe works and what to look for when porting. Overall, I would say that the procedure is not very complex, but it does take a significant amount of effort for each application to port, even after getting past the steep learning curve, simply because of the lengthy procedure required to port an app.

One of the most difficult aspects to deal with was the sheer amount of time it takes for some applications to be analysed. For example, in the case of the Minecraft server, the Loupe analysis took approximately 3 hours, because of the length of time needed for the server to initialise and start. Although this was the longest example that I had to port, other server applications routinely take 10 to 15 minutes to complete an analysis, and if the results are invalid or the test throws an error, that means another 10 to 15 minutes added to the porting time.

However, from my experience, the most difficult aspect when porting an application was not the fault of Loupe, but stemmed from the lack of documentation for many of the applications that I had to go through. Excluding the applications that were completely dead, such as cado or deno, which was completely deprecated and required an auxiliary file to work which was nowhere to be found, even in the case of applications that were otherwise simple to port, the documentation, or lack thereof, was the main factor that added time and effort on my part.

Throughout the first semester, myself and another Loupe developer have analysed over 100 applications from the OSv application list. This was very useful because we gathered a lot of information about Loupe, where it shines and where it can be improved. However, the given number also includes some applications that were impossible to port (which will be explained in section 3.3.). The total number of applications that were successfully ported ended up being 62. As more development efforts go into Loupe, I am certain that the percentage of applications that are impossible to port will steadily decline. At the same time, the proportion

of applications that were not able to be ported will also change if we consider a different set of applications. These results complete the first aim in my list.

3.1.4 Visualising the Data

After obtaining the relevant application list, it was time to put this data in context. Even if the data would not show a significant difference in using the optimal support plan generated by Loupe, we were expecting it to at least highlight the benefits of stubbing and faking system calls when porting an application to a new operating system. There was one way of finding out. My goal was to create two charts, showing the number of applications supported for every system call that is implemented in the case of the natural, “organic” development order of the OSv applications, and the case of the OSv developers following a support plan generated by Loupe. To help me out, I wrote some programs in Python which gather and translate the data to numbers that can be plotted onto charts by doing the following:

1. Retrieve the OSv application list in chronological order using the Git API and output them as a CSV file.
2. Take the resulting CSV file and, using the Loupedb folder, create the most optimal way of implementing, stubbing and faking applications given their order, both in reader friendly and spreadsheet friendly outputs.
3. Take the result from the Loupe support plan and convert it into a spreadsheet friendly output.
4. Convert the data to flip the X and Y axes (instead of numbering the system calls to the applications, it numbers the applications to the system calls)

All of the scripts are included in the project files. Although we were preparing for the worst case scenario, we were pleasantly surprised by the results, shown in the following figures.

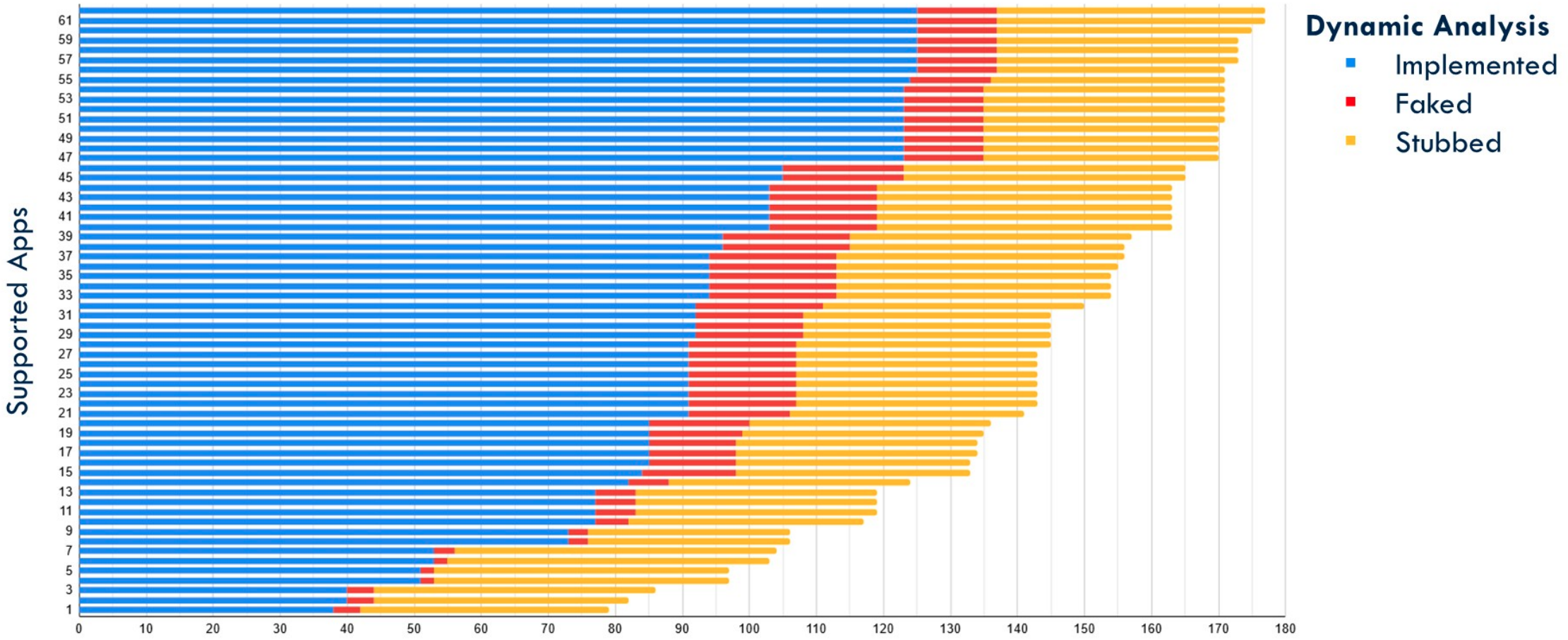


Figure 3: The natural development of OSv applications, assuming stubbing and faking was used.

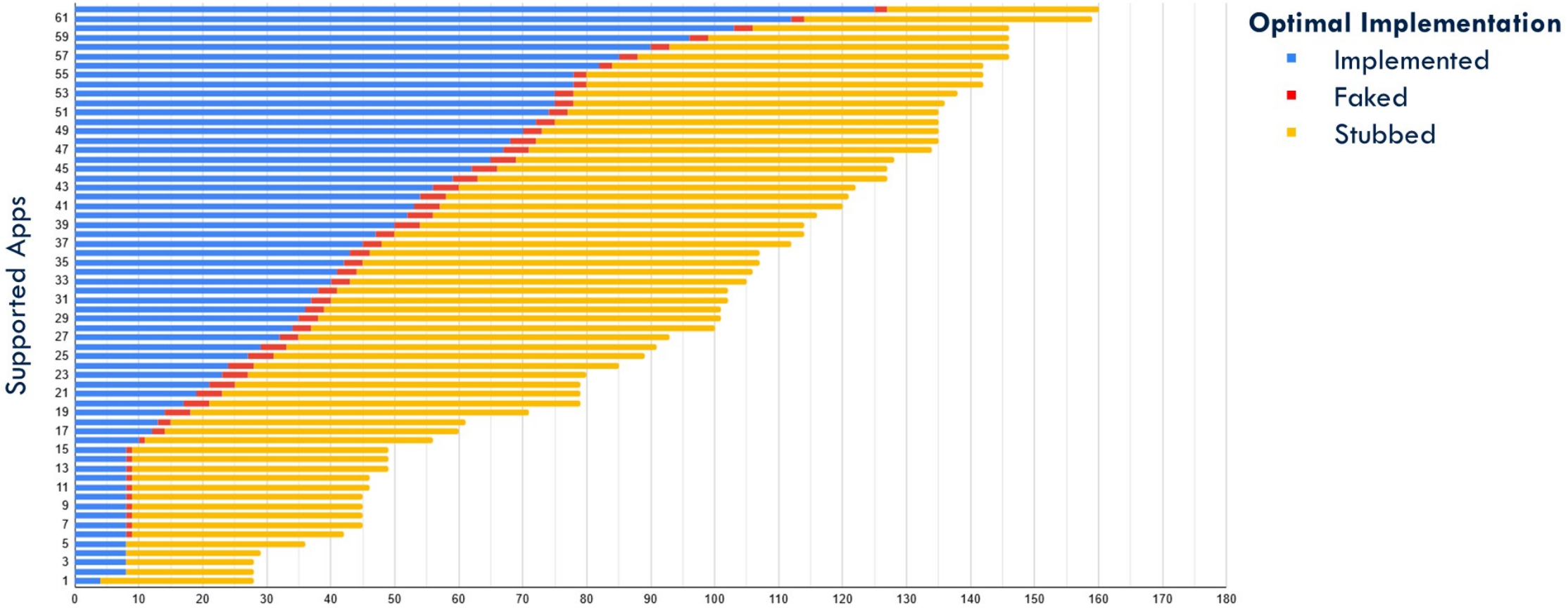


Figure 4: The optimal development of OSv applications according to the Loupe support plan.

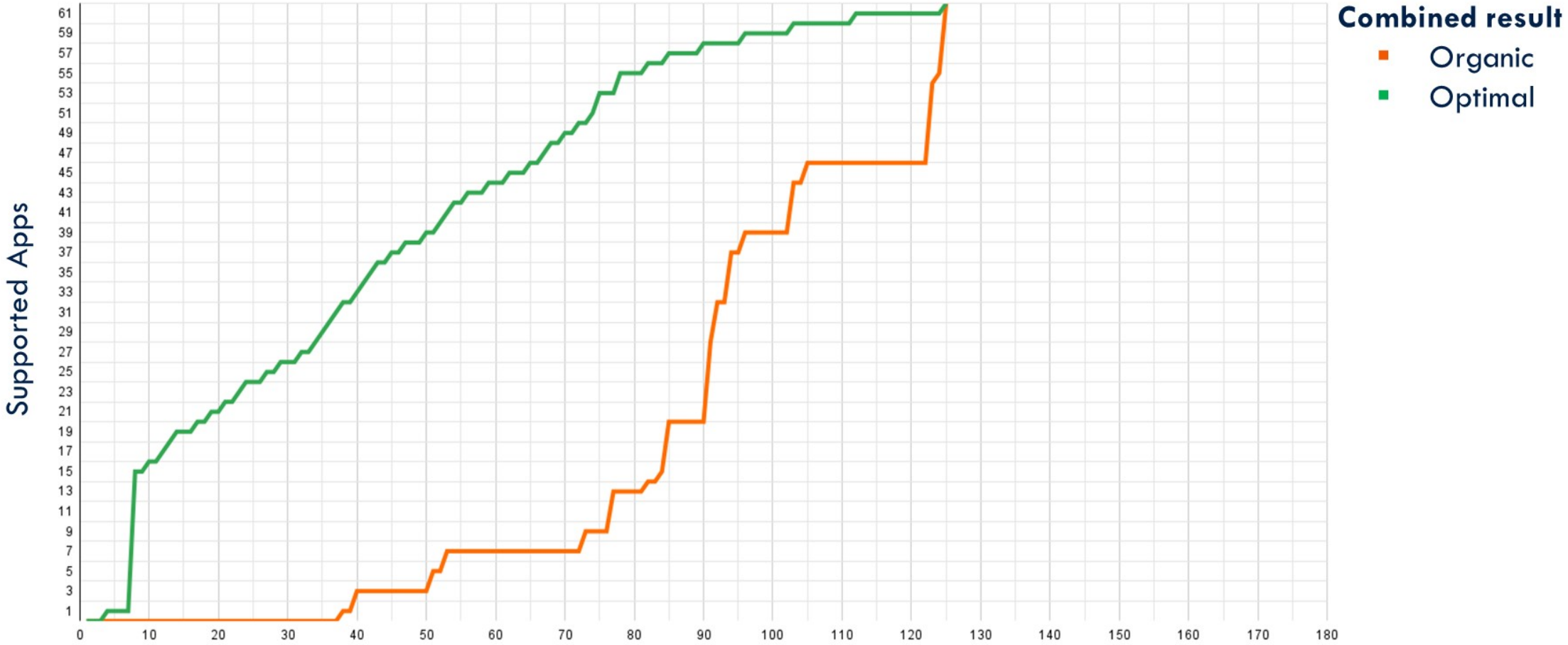


Figure 5: Overlaid charts for both cases: the “organic” order with orange, and the optimal way with green.

3.1.5 The Impact of Loupe

As mentioned earlier, an operating system is only as good as the applications it supports. This is doubly important for operating systems that are in development. Although the end result of the previous charts is the same (which is to be expected, the application list is the same, and it consequently requires the same system calls to function), the true impact is revealed during the process.

Usually, when a developer wants to create an operating system, he chooses an app, implements all of the system calls for it to work, and then moves to the next app. While this approach can be good for the enjoyment of a project, the developers need to put in a significant amount of effort in the beginning until they can achieve a respectable number of applications. The resulting development curve is very steep in the beginning, and after a certain point there are a lot of big jumps. The Loupe support plan works the other way around, having no plateaus like the organic development, and the development curve only steepening towards the end of the application list. If the aforementioned developers were to follow the support plan by Loupe, they would still reach their target application list, but they would see a very constant return on their efforts.

Putting this in the context of OSv, and assuming that the developers stub and fake system calls where appropriate, and that every system call takes roughly the same amount of development time (which, of course, is not the case), the OSv developers would have reached 20 applications supported with almost half of the development efforts. If we only count the system calls that have been actually implemented, the developers need to spend five times as much development time and effort for the first 20 applications. In a scientific conference, this can mean the difference between the project being rejected and it receiving the initial funding needed to surpass the initial development hurdles.

3.2 Contributions to Loupe

Having had the opportunity to work on porting the applications with the help of Loupe, I had some ideas as to how I could make the tool work better. This varied from minor quality of life

fixes, to more major features that I could implement. This section will present my own improvements on the tool.

3.2.1 Quality of Life Features

While working on the application list, I noticed that the process of porting applications has the potential to be streamlined. For instance, every time an application was analysed, the entire “Dockerfile_data” folder was copied to the “loupedb” folder. At the same time, after this was done, the user had to manually look in the resulting folder, to see if the test was valid. Both of these steps can be handled by Loupe instead of the user.

Thus, I added new features to address both of these points. In the case of the first one, I added a function that analyses the Docker file to see what folders were copied, created a list with the relevant files and only copied the files in that list. In the case of the second one, I added a function that analyses the CSV file created by Loupe in order to determine if the result is invalid or not. On invalid results (such as all of the system calls can be stubbed and faked or none of the system calls can be stubbed or faked), the function prints an error message and stops the program.

Another aspect that I noticed was that in the case of some applications, the static analysis was causing problems when the dynamic analysis may have produced valid results. Therefore, I added a separate parameter in the explore.py program to disable the static analysis of the applications.

3.2.2 The Popularity Feature

As was shown earlier, one of the most important features of Loupe is its ability to create the support plan, in order to provide the most optimal way of implementing, stubbing and faking system calls for the highest number of applications supported. However, this assumes that the operating system developer already has an application list in mind, which is not always the case. For operating system developers, the ultimate goal of their project is to support as many applications as possible. Thus, Loupe needs to also offer a general priority list for all of the system calls, that would not need a pre-existing application list.

The solution I decided to implement is a general popularity feature for the system calls. This feature analyses the system call usage of all of the applications in the provided database folder and orders the system calls in descending order of the percentage of applications that use it. I have also used this feature for the next part of the project.

3.3 Limitations and Challenges

Although Loupe is a tool that works for almost every app, it does not come without its limitations. As was shown in the previous experiment, out of 100 applications analysed, only 62 applications ended up being ported. However, it is important to mention that part of the applications from the OSv application list were either duplicates, deprecated applications, dead applications or were not applications at all. At the same time, the impossibility of porting for some applications stemmed from the drawbacks of Loupe. Some of these come from Loupe's design, while others are under development and should be addressed in the near future.

3.3.1 Reliance on Docker Containers

The most important issue is the tool's reliance on Docker containers. Because of this reliance when analysing an app, it makes some applications impossible to port. One such example is "xapps", which can not work in a Docker container because it requires graphical support, or "jruby-sinatra", which can not be installed on Docker. Other applications that need to access the kernel can not work in a container and therefore can not be analysed by Loupe. This limitation is the hardest one to address, as a lot of the tool's functionalities rely on containers, but at the same time the number of applications that do not work because of this is relatively small (at least with the applications that I analysed)

3.3.2 Complexity

Another limitation of Loupe comes from the way it analyses an app. As mentioned previously, the user needs to specify the exact binary which will be analysed by the secondary "explore.py" program. This works well for most applications, but for other applications, such as "apache-spark", porting becomes very complicated at best, and impossible at worst. This is because such applications do not have a main binary, instead relying on many wrapper scripts

to execute its features. By selecting a single binary, we are not actually testing for the app, just for the central component, which may not work at all on its own. This is a feature that the developers acknowledge and are planning to mitigate in the future.

3.3.3 Deprecated Applications

The last limitation that I will mention with Loupe only applies in the context of the application list that I worked with. A person trying to port an application needs to make it work in a Docker container. As with the previous limitations, this is fine for most applications, but in the case of the OSv application list, a lot of the applications no longer had any support, or relied too much on other applications to work properly. Other applications have very limited or no scripting support, and very poor documentation as is the case with "lfe"; this makes the porting process very complex, especially if it is done by someone other than the application developers. This, of course, is not something that the Loupe developers can feasibly mitigate, but this problem should not occur in the case of more popular and well maintained applications.

3.3.4 Getting Used with Loupe

Starting my work for this project, I did not have a lot of experience with anything related to the tools that I used. I had some experience with C and Python, but I did not know how Loupe or unikernels worked. Therefore, a lot of my time in the first semester was spent getting used to using Loupe and porting an app. The fact that Loupe was in an early stage of development also meant that the documentation was sparse in certain places, and for those cases my best bet was to look through hundreds of lines of source code or try and adapt pre-existing examples. Eventually, I got used to my environment, and the problem with documenting should be solved in time.

4 Test-Driven Unikraft

4.1 Unikraft Test Suite

In the second semester, after learning a lot about application compatibility, I decided to apply it to a specific context. This lead me to focus on Unikraft, and more specifically on their system calls. The way that they implemented the system calls was in a largely case by case basis, implementing system calls in order for applications to work and adding onto them upon encountering a bug. While this might be suitable for some more basic system calls, such as read or write, it is not enough for more complex system calls such as mmap or fcntl, which have a range of uses and functionalities. Thus, the best solution for this was to devise a test suite for the system calls. Fortunately, the Linux Test Project (abbreviated to LTP) provides a test suite for the entire Linux kernel, including the system calls. Consequently, I came up with two possible solutions for the test suite, each with their pros and cons.

4.1.1 Solutions

The first solution consists of creating a Unikraft application that runs the entire LTP. This was also the first solution that I attempted. However, this proved to be unfeasible, for three reasons: firstly, one of the steps of creating a new application involved getting the gcc arguments and transcribing them to a makefile. This is fairly simple to do for smaller programs, such as the example they give with iperf. However, for a larger program with a lot of scripts and smaller programs such as the entire LTP, gcc outputs so many arguments that they constitute megabytes of text. Secondly, the LTP can not work on a system without first building it, which in turn generates the “config.h” file. This is standard practice for a lot of applications, but in the case of the LTP, the file was very complex, with a lot of the features being specific to my host system and not Unikraft. It would take too much time to see if the tests did not work because of the badly configured file and not the operating system itself. Thirdly, the entire project uses tests for every aspect of the Linux kernel, as well as some related features. This means that a large part of the test suite would not work for Unikraft by design, meaning that a lot of the effort would be wasted.

The second solution entails taking the system call tests in particular and adapting them to function in a Unikraft image. I decided to go with this approach because it is more scalable

and focused than the first solution, resulting in a more efficient and result-oriented use of the limited developing time that was left. At the same time, I can make use of Unikraft's newest feature: the ELF (abbreviated from "Executable and Linkable Format" [28]) loader. This feature can run any application inside the Unikraft environment by loading the binary of the application, without the need of creating a new application for Unikraft, by loading the machine code as given by the binary into the memory, thus making it executable by Unikraft. With this feature, a test can be compiled, and then the resulting binary can be sent to a Unikraft image and use the ELF loader, having the potential to be automated. So, the task of developing the test suite consisted of three main steps: make the tests work outside the LTP framework, send the resulting binaries to the ELF loader, which will execute the tests, and then analyse the results and send them to the user.

The best part about this solution is that it also applies in many more contexts than just Unikraft. Indeed, because all of the test files will be turned into stand-alone C programs, it works in any environment or prototype operating system capable of executing compiled C code.

4.1.2 Overview of Unikraft

Because Unikraft is such a central part of my task, I had to get familiarised with the ecosystem before doing anything else. With unikernels being such a novel concept to me, I was expecting to encounter a steep learning curve, and while this would certainly be the case for other unikernels, Unikraft provides a lot of support, both on their website through the wiki and on their Discord server.

In order for a user to set up an existing app, there are two options available. The first one is to use the provided command line helper tool "kraft", which simplifies the process of setting up a Unikraft image to just a couple command lines. This tool is especially useful for applications that have already been set up and are part of their ecosystem, but, as will be exemplified later, it does not work for every application out of the box. The second option is to use a makefile, which requires more effort from the user, but is balanced by the user's familiarity with this system. For more complex applications fetched using the make-based

approach, the user must manually pull and set up any library dependencies that the application might need. Figure 6 shows how Unikraft is structured.

```
|-- apps - This is where you would normally place existing app builds
|-- archs - Here we place our custom arch's files
|-- libs - This is where the build system looks for external library pool sources
|-- plats - The files for our custom plats are placed here
`-- unikraft - The core source code of the Unikraft Unikernel
```

Figure 6: The basic folder structure of Unikraft [6]

When it comes to setting up a new app, however, the process becomes much more complex. As will be presented in the next sections, this was one of the major challenges that I had to overcome.

4.1.3 Partial Automation

The first step was to analyse a few system call tests and determine the difficulty of the task. All of the tests for the system calls read, write, open and close were analysed. The tests were compiled and executed on a Linux system before testing in the Unikraft environment. After this, a partial automation script was devised, in order for it to take the remaining tests and streamline the process of porting by replacing as many of the common steps as possible.

The script performs the following steps:

1. Remove all of the comments, first the single line and then the multiple line comments. This was done to make the resulting code easier to go through.
2. If there is no main function, add one. Most tests do not have a main function, which makes them impossible to compile and run on their own.
3. Convert the verify function to an int type instead of void. The typical test has a verify function, which runs the test. In the ported version, the verify functions are made to return 1 if the test passed, 0 if it failed and -1 if there is an error.
4. Automate other LTP specific strings. This can include replacing the test function of the system call with assigning the function to a variable, replacing the test output function with the printf() function from the stdio library and other test specific strings (which will be explored further in section 4.1.5).

This worked wonders for a lot of tests, making them work with minimal effort. For others, more effort was required to make them compile and run successfully. Some tests proved to be impossible to make work, as they were either too reliant on the LTP framework or were simply not made to be compiled and executed on their own. This aspect will be expanded further in section 4.3.

4.1.4 From Semi-automated to Fully Functional

After step one was complete, all that was left to do is to take the remaining tests and automate them. Because of the limited development time remaining and the large number of tests (1206), only a small but very significant portion of the system calls were finished. The importance of the system calls was determined using the previously mentioned popularity feature of Loupe. A total of 201 tests have been considered, from 105 system calls, including the 30 most popular system calls in the list. This number may be small in comparison to the total number of tests, but this selection of system calls represents the majority of the system calls that a typical application would require. According to figure 7, system call significance has a very steep decline early on. In fact, if we only consider the set of 30 most popular system calls, they represent approximately 95% of the typical selection required by an app.

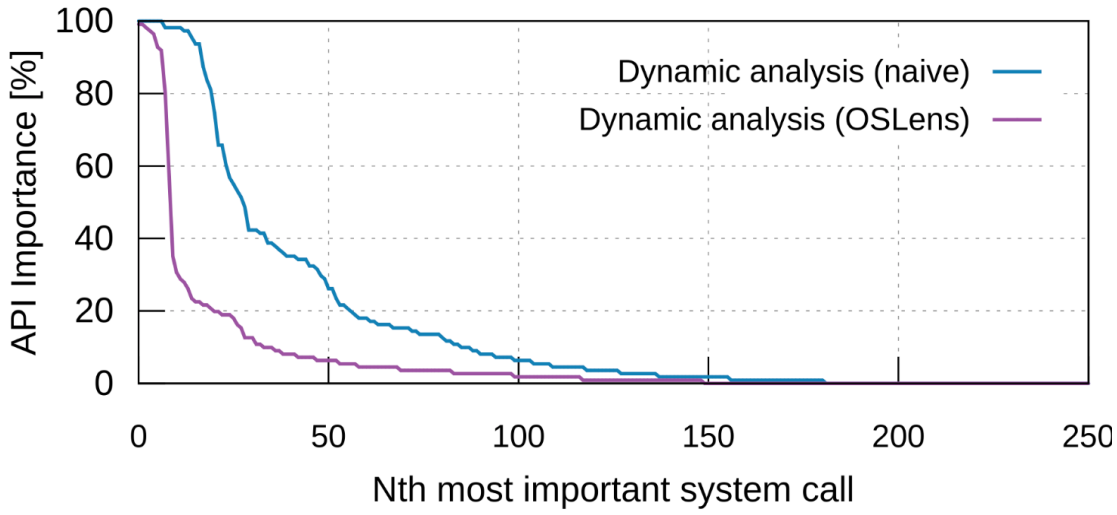


Figure 7: A chart presenting the significance of system calls for a random app. OSLens is an alias for Loupe used to anonymise the tool

4.1.5 Extrapolating

The task of adapting all of the system call tests was not feasible to complete in the given development time, even with the use of partial automation. Thus, after working on a significant and relevant portion of the tests, a general methodology for adapting them was formulated. For instance, many of the LTP functions have an equivalent in the standard C libraries. In the case of system calls, a lot of them are very straightforward (for instance, the LTP SAFE_READ() function can be replaced with the standard read() function). Other system calls can be used with the “syscall.h” library, with the appropriate system call number. Test files also have a fairly standard structure. Usually, there are 4 functions in the test: the main function, a setup function, a clean-up function, and a function that actually verifies the output. This structure can vary slightly on a case by case basis, which is the reason for me not being able to automatically adapt all of the test files.

I used the visual studio code’s intellisense for undefined functions or constants to figure out the libraries that I needed to include and the functions that I needed to replace. For this reason, my project file also includes the original test files from LTP.

In summary, what the user needs to do, other than dealing with the case by case variations of each test, is the following:

1. Choose the libraries to include. For most cases this includes stdio, errno, and fcntl among others, but other test cases may include a lot more.
2. Replace the LTP functions with the standard C functions. Most of the time, this can be done fairly reliably, although from my experience, this is the step that I spent most of my time on.
3. Make sure the automation prerequisites are met. They will be presented in more detail in subsequent sections.
4. When in doubt, check the original test files. Sometimes, the partial automation script may malfunction, removing otherwise relevant lines of code. Other times there may be a problem with solving step 2. In any case, if the provided original test files are not useful, the files can also be found on the LTP GitHub repository, under “testcases/kernel/syscalls”

4.2 Central Module

With some of the most important tests being ported, and with all others being a basis for a good testing framework, it was time to automate the testing process, and create a central module that would tie everything together. Most of the automation requirements were set in place previously, and they already account for a lot of potential variables.

4.2.1 Docker Environment for Unikraft

In order to have as few parts that could give errors as possible, I decided to run the test suite through the use of Docker containers. The main advantage of this usage is that a developer does not need any prerequisites (other than Docker) to run the test suite. At the same time, the environment will always be constant, no matter which system runs it. This is especially important for a diverse team of developers such as the Unikraft developers, which use a variety of operating systems and architectures to work on Unikraft. Additionally, some of the tests are specifically made to break your system (as is mentioned in the LTP GitHub repository and their website [25]). To give a more concrete example, one of the tests for “mmap” tries to replicate a bug that occurs when the entire partition is full. By using a Docker container, there’s an extra layer of protection against what could potentially break the host environment.

For the Docker container, the first step was to make Unikraft work in the environment. This was not too difficult to set up, as the Unikraft developers already made a Docker container work with their helper tool, “kraft” [29]. This was perfect for installing any applications or libraries that are already used by Unikraft. However, because the ELF loader works as an application, and because it is still early in development, it does not work with kraft by default. The alternative would have been to work with the makefile menu. In any other setting this would not have been a problem, but in the case of a Docker container this was not an option, because it needs a graphical interface to work. My other alternative was to exploit the way that kraft works, by manually writing the appropriate configuration files necessary for it to work after cloning the ELF loader from GitHub. This proved to be effective, resulting in a fully functional Unikraft environment in the Docker container.

4.2.2 Automation Prerequisites

In order for the test suite to work properly, two conditions must be satisfied. The first requirement only applies to system calls that have more than one test file. In that case, the user needs to create a header file, with all of the required libraries for applications. In the partial automation script, the programs assume this header file to be named “incl.h”. I decided to do this because I noticed that for the same system call, each test file has similar and overarching libraries that need to be included.

The second requirement is that the test needs to output “test succeeded” in the case of a passing test. This can be written alongside the rest of the test output. I decided to do this because the test outputs vary widely across system calls, and for the automation to work, part of the output needs to be standardised.

My test suite will still run if these prerequisites are not met, however the results will be inaccurate. If the first condition is not met, the tests will figure either as failing or as unfinished. If the second requirement is not fulfilled, passing tests will figure as failing.

4.2.3 Loading the Test Binaries

With this step done, it was time to load the tests in the container and make them work in the Unikraft image. The ELF loader works in two types of executables: static-pie executables and dynamically linked binaries. The main difference between the two is that the first format is static, meaning that the machine code will be executed, regardless of where it is placed in the memory [30], whereas the second format needs to be linked to the respective dependencies to be correctly executed. Consequently, the procedure for static executables is much simpler and reliable, with the user only needing a single command line pointing to the binary in order for Unikraft to execute it. This approach is also the most stable of the two, in the case of the ELF loader. In spite of this, in order for it to work, the user often has to recompile the application in a static-pie format, which is not something that developers might want or be able to do. Extra steps from developers invariably means that less applications will work for Unikraft, therefore this approach is used sparsely, for applications that are already compiled in this format. This, however, was not a problem for my tests, as I could compile them in the static-pie format. What was a problem was the fact that it still did not work in the Unikraft

environment, with every test displaying unexpected outputs that could not be analysed by my program.

Secondly, there was the dynamically linked executable. This method is a lot more complex and a lot less stable than the previous one. This option, as opposed to the previous one, required that I install and configure extra libraries, as well as mounting a shared 9p filesystem on the image. In the shared folder, I needed to include the library prerequisites alongside the binary, and use a different command when running the image. Luckily, since all of the test cases are simple C programs, using no third party libraries, all of the tests had the same dependency requirement and needed the same command to run, meaning that it would be much easier for me to automate the test suite. However, try as I might, this feature proved to be too unstable to work at the moment. In fact, after contacting the Unikraft developers, they said that this feature is still under development, with it not even working on their provided hello world example. Thus, I was at an impasse. Because this is a problem that is outside the scope of this project, and with my limited development time, I decided to compile and run the tests directly in the Docker container as a placeholder. The overall automation process is unchanged, because the Unikraft environment and the test suite is set up. When the developers will finish the ELF loader, all that needs to be done is for one line of code to be changed and it will be fully functional. Initially, I intended to make it as simple as possible, with one parameter that can change between the placeholder execution of the tests and the actual environment. However, because the ELF loader feature was still in development and changing rapidly, this could also make my program malfunction and give incorrect test results. I decided to change this by hand when the feature will be implemented.

4.2.4 Master Program

So far, I presented the way that the Docker container functions. This section will present the master program, which runs and compiles everything, as well as providing the user log files and a visual representation of the test result as an optional parameter. What it does is it builds and runs the Dockerfile. In order for it to work, the user needs to provide a GitHub access token as a parameter, a requirement of Unikraft's helper tool. Afterwards, it has a set of optional parameters, one that enables the creation of log files for each test that fails, and one that displays all of the test results in a series of plots. This program was written in Python,

making use of the “subprocess” and “os” libraries for the execution of tests and creation of log files, and matplotlib and numpy for the visual representation. Figure 8 shows an example of the graphical output.

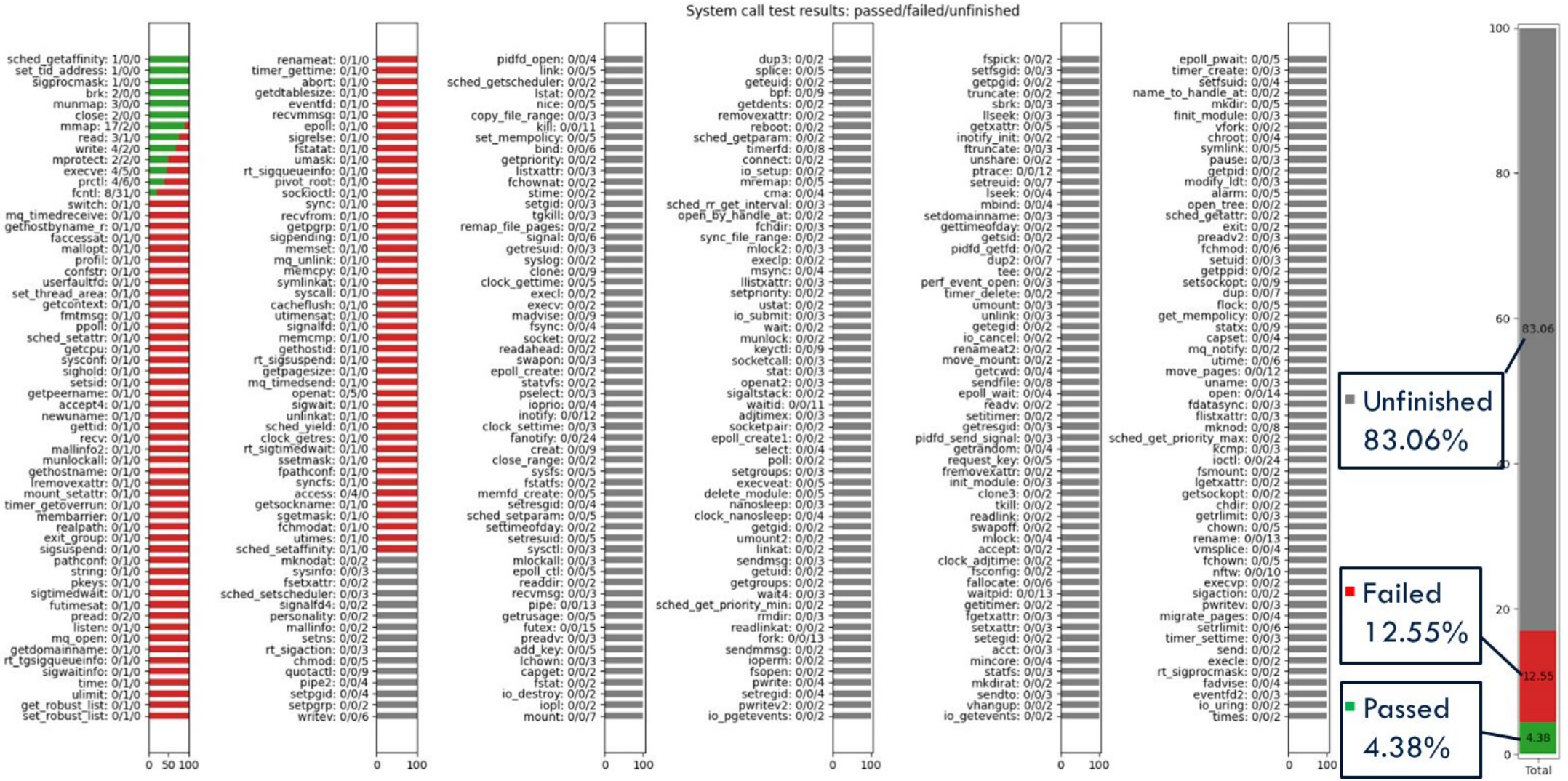


Figure 8: The graphical output of the master program

The graphical output consists of two parts: the first six plots show the test results for each system call, and the right-most plot shows the overall progress of system calls, with percentages written on each section. Tests that pass are coloured in green, tests that fail are coloured in red, and tests that are broken or unfinished are coloured with grey. For better readability, the tests are grouped, with unfinished tests being last, and the rest of them being sorted in terms of the highest percentage of passing tests, as shown below.

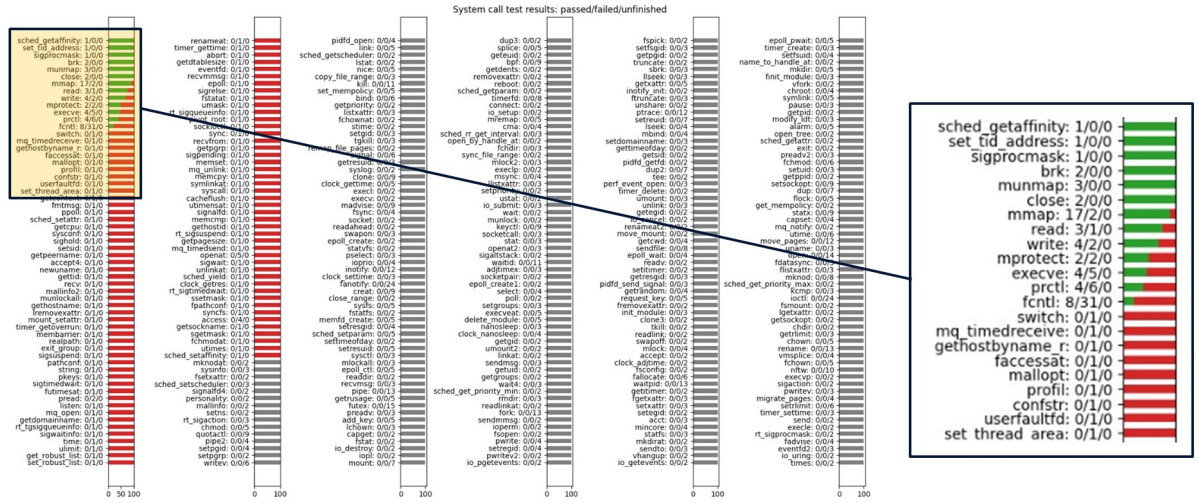


Figure 9: The graphical output, focused on a part of the system call results.

The interface also provides exact numbers for each system call, in the format of “passing tests / failing tests / unfinished tests”. Lastly, the program provides a text output with the total number of tests.

4.3 Limitations and Challenges

Although the chosen solution answers a lot of the problems raised by the alternative presented in section 4.1, it is not without its flaws. Compared with the first semester, where the main challenges that I had to face were related to the theory, this semester’s challenges were a lot more applied in nature. I believe this to be a personal success, as it highlights my progress throughout the year. This section will present the challenges that I had to face, as well as the current limitations of my implementation.

4.3.1 Docker Containers

Just like the limitations in Loupe, Docker containers represent both a boon and a detriment. After making Unikraft work in the Docker container, and compiling the tests, some of the test cases that gave a good result or compiled successfully no longer work in the container, such as test 4 for the “open” system call. Some of these errors can be mitigated by running the container in privileged mode, however this has some obvious security implications.

4.3.2 Reliance on the ELF Loader

Because this test suite relies on the ELF loader, a component that is still under development and unstable, some of the tests that fail might do so due to a fault in the ELF loader, as opposed to the component that the test is probing. Because of this, some of the test results might not be as accurate with this alternative.

4.3.3 The LTP Framework

By far the biggest challenge to this semester’s work was understanding the LTP framework, from setting up the project, to understanding what each test does, to understanding the various proprietary functions. LTP is a very complex project, and represents years of work done by a multitude of experienced software engineers. Due to my previous experience with C, understanding and debugging the programs themselves was not that big of an issue, but when it comes to adapting the tests, I had to spend some time trying to understand how to make them work. Fortunately, I managed to overcome this obstacle.

4.3.4 Automation

Due to the size of LTP, it was imperative that I use automation to speed up my tasks' completion, especially when this task includes a lot of repetitive steps that can be accounted for. That being said, automation is not the answer to everything, as it does not account for more special test programs, which need to be compiled and run with certain parameters in order for them to pass. On a case by case basis, this was the most time consuming part of porting the tests.

5 Conclusions and Recommendations

Since I am studying the Human Computer Interaction course, my programming module selection was limited compared to other Computer Science courses, especially when it comes to low levels of abstraction such as operating systems, processor architecture and even engineering. Therefore, when I first started working on this project, I barely had any knowledge other than my own personal endeavours. This project has given me a much better understanding of operating systems in general, and I think it balances out a lot of the areas that I missed in my previous years.

5.1 Evaluating the Aims

I believe that, throughout this year, I have achieved every point that I set out. For the first part, I was able to carry out an experiment that has been integrated into a larger paper. This paper has now been submitted to ASPLOS, which is an A* ranked systems software conference. This thoroughly proves my achievements of both the first and the second points, as shown in the table in section 1.2.

In the second part, I managed to get familiar with both Unikraft and the LTP. Although I was not able to make the tests run inside the Unikraft environment, I was able to set up a working Unikraft environment inside a Docker container, as well as making the command line tool “kraft” work for an app that was not configured. My work regarding the test suite has been presented on the Unikraft Discord server and has attracted the attention of the community.

The visualisation of the data that I produced for both parts prove my achievement of the third aim. The data is concise and easy to understand, both in the case of the charts that I conceived, and in the case of the graphical output of the test suite.

5.2 Conclusion

This project was an incredible learning experience for me, and I thoroughly enjoyed the time that I spent learning for it. Although there were ups and downs, and the project got repetitive at times, I feel proud and satisfied with the results that I achieved and the knowledge and

experience that I gained. Furthermore, I have achieved even more than I originally set out to do, having contributed to Loupe and performed the experiment in the first part of the project. Throughout this year I learned how to use Docker containers and how containers work, going from knowing nothing about Docker to being able to comfortably use them in my own projects.

Before starting this project, the only tangential experience that I had was being able to use QEMU to create virtual machines, but now I learned a lot about compatibility technologies, as well as Loupe and its impact. The experiment that I completed is going to be extremely helpful for the Loupe and Unikraft developers, and the contributions that I added to Loupe have streamlined the process of application porting.

The second part of the project helped me understand the importance of tests in almost any context, and made me better appreciate the difficulty of porting an application for a different operating system. I learned about Unikraft and how it works, and I managed to create a framework for these tests which will no doubt be useful for further developing the operating system. At the same time, this project has helped me appreciate the importance of documentation for even the smallest of projects. This was a major obstacle that I had to surpass, both in the first and the second parts of the project.

5.3 Further Work

This project, however complex, still had a relatively short development period. This section will present ideas for future work regarding this project.

5.3.1 Expanding the Work

The first action that I would do is to keep doing the smaller activities that I performed in both parts of the project. For the first part, this would mean performing another experiment with a higher number of applications, allowing for more conclusive results with a higher precision.
For the second part, this translates to porting all of the system call tests, which would give operating system developers a much clearer picture of their progress.

5.3.2 Investigating the Difference between the Organic and Optimal Plans

One aspect that I noticed when I visualised the results is that, although the number of implemented system calls is the same, the stubbed and faked system calls show a higher number in the case of the organic order. I checked my programs for bugs and I could find nothing, as far as I know the data is entirely valid. This aspect should be checked further, either by double checking the Loupe support program function or to try and find a reasonable explanation for this phenomenon.

5.3.3 Creating a Separate Test Framework

A lot of the limitations for the second part were caused by the tests being taken from LTP. In an ideal scenario, I would like to create a system call test suite that would be able to work for the majority of prototype operating systems. Because the LTP tests for many different aspects of the Linux kernel, as opposed to just the system calls, my proposed test suite would only test for system calls.

This test suite could also be split into different parts for more complex system calls, based on the different types of functionalities. Ideally the test suite would not require any proprietary functions or include parameters, only relying on standard C.

The central module could also be made more modular, with a separate script responsible for the setting up and loading of tests for every potential operating system that would benefit from this test suite.

5.3.4 Making the Visual Output Interactive

Currently, the visual output for the test suite is a collection of graphs. My idea would be to make the interface interactive. Since the central program already offers log files for each test, the module could be made so as to display the log when hovering over a failed test. Another example would be the option for the user to customise the system call list, in order to focus on a certain subset of system calls without cluttering the screen, or to customise the colours of the tests. These features were not implemented because I considered that they were not as relevant for the scope of my current project and therefore I could use my time better elsewhere.

5.3.5 Make a More Substantial Contribution to Loupe

As was detailed previously, I had the privilege of contributing to Loupe. However, my contributions were relatively minor. While I still have more ideas for minor fixes, it would be interesting to contribute to Loupe in a more substantial way. One good example entails revising the differentiation between suite tests and benchmark tests. Currently, when asking for a support plan and asking for all of the applications and all of the workloads, the command line is slightly misleading, because it generates a support plan that only includes the applications that have both the benchmark and the suite test results, instead of including all of the applications as expected. After analysing a little further, I noticed that a lot of the suite tests were hard coded into the Loupe program. I would like to investigate this further and potentially fix this.

5.3.6 Find a Way to Automate Application Making for Unikraft

One of the most difficult actions for Unikraft is the process of setting up a new application for Unikraft. It would be very interesting to automate this process, especially for open source applications (at least partially). For other applications that need to be installed, the hypothetical program would scan the make file and find certain requirements. Doing this would allow Unikraft’s application roster to skyrocket, which would greatly help the popularity of this operating system.

6 Bibliography

1. *Hypervisors vs. Lightweight Virtualization: A performance ... - IEEE xplore*. Available at: https://ieeexplore.ieee.org/abstract/document/7092949 (Accessed: April 21, 2023).
2. *KVM. Available at: https://www.linux-kvm.org/page/Main_Page (Accessed: April 21, 2023).*
3. *QEMU*. Available at: https://www.qemu.org/ (Accessed: April 21, 2023).
4. *Accelerated, containerized application development* (2023) *Docker*. Available at: https://www.docker.com/ (Accessed: April 21, 2023).
5. Cambridge, A.M.U.of *et al.* (2013) *Unikernels: Library Operating Systems for the cloud: ACM SIGARCH Computer Architecture News: Vol 41, no 1, ACM SIGARCH Computer Architecture News*. Available at: https://dl.acm.org/doi/abs/10.1145/2490301.2451167 (Accessed: April 21, 2023).
6. *Unikraft*. Unikraft. Available at: https://unikraft.org/ (Accessed: April 21, 2023).
7. Simon Kuenzer NEC Laboratories Europe GmbH *et al.* (2021) *Unikraft: Proceedings of the sixteenth european conference on computer systems, ACM Conferences*. Available at: https://dl.acm.org/doi/10.1145/3447786.3456248 (Accessed: April 21, 2023).
8. *Macworld 9304 April 1993 : Free Download, borrow, and streaming* (1993) *Internet Archive*. Available at: https://archive.org/details/MacWorld_9304_April_1993/page/n109/mode/2up?view=theater (Accessed: April 21, 2023).
9. *RetroArch*. Available at: https://www.retroarch.com/ (Accessed: April 21, 2023).
10. *EmulationStation*. Available at: https://emulationstation.org/ (Accessed: April 21, 2023).
11. *Chapter 11. linux binary compatibility* (no date) *FreeBSD Documentation Portal*. Available at: https://docs.freebsd.org/en/books/handbook/linuxemu/ (Accessed: April 21, 2023).
12. *What is wine?* (no date) *WineHQ*. Available at: https://www.winehq.org/ (Accessed: April 21, 2023).

13. ValveSoftware (no date) *ValveSoftware/Proton: Compatibility Tool for steam play based on wine and additional components*, *GitHub*. Available at: https://github.com/ValveSoftware/Proton (Accessed: April 21, 2023).
14. *Strace* (no date) *strace*. Available at: https://strace.io/ (Accessed: April 21, 2023).
15. *Unikraft and the coming of age of Unikernels* (2023) *USENIX*. Available at: https://www.usenix.org/publications/loginonline/unikraft-and-coming-age-unikernels (Accessed: April 21, 2023).
16. *Unikernels: Rise of the Virtual Library Operating System* (no date) *Unikernels: Rise of the Virtual Library Operating System - ACM Queue*. Available at: https://queue.acm.org/detail.cfm?id=2566628 (Accessed: April 21, 2023).
17. *The operating system designed for the cloud* (no date) *OSv*. Available at: https://osv.io/ (Accessed: April 21, 2023).
18. *HermiTux* (no date) *Hermitux*. Available at: https://ssrg-vt.github.io/hermitux/ (Accessed: April 21, 2023).
19. Takaya Saeki The University of Tokyo *et al.* (2020) *A robust and flexible operating system compatibility architecture: Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, *ACM Conferences*. Available at: https://dl.acm.org/doi/abs/10.1145/3381052.3381327 (Accessed: April 21, 2023).
20. *A comparison of virtualization technologies for HPC | IEEE conference ...* (no date). Available at: https://ieeexplore.ieee.org/abstract/document/4482796/ (Accessed: April 21, 2023).
21. Bagherzadeh, M. *et al.* (2017) *Analyzing a decade of linux system calls - empirical software engineering*, *SpringerLink*. Springer US. Available at: https://link.springer.com/article/10.1007/s10664-017-9551-z (Accessed: April 21, 2023).
22. Tchakounté, and Dayang (no date) *System Calls Analysis of Malwares on Android*. Available at: https://d1wqtxts1xzle7.cloudfront.net/52584050/5513797455-libre.pdf?1491934087=&response-content-disposition=inline%3B+filename%3DSystem_Calls_Analysis_of_Malwares_on_And.pdf&Expires=1682076483&Signature=ZGVVSLA1tmRkqMbaKkIK49STFKXr~5rmRuvx3tIhde5ugHZRuWGByIRSuWBUeLFcvSrddtr1~jEIpQcAyKg7q-

K3tXLC8dWoSFgZJiAI8kbwnEWmga4V7GtVLpt7hXE5lPbRsrEfh1Rfgd~VxxEApVFvBRiwbJ0aQFYJBAxFm168R3oh7wcNmYCk8pbrdzF00Lmv1Gp1zrMqSHKGXCj6lMheELC-b9xCVKWHOmzkEa-Husp5io8BVE9OhLy-zFgepiq9xIw~jzxTR8AwvhmQwmthU5ra9I~AAK0mE91myV4LRh663WRowkU02H2yw0uTzT9cnjCs9~Ytxc7J9blxMQ__&Key-Pair-Id=APKAJLOHF5GGSLRBV4ZA (Accessed: April 21, 2023).

23. *Testing Linux with the Linux Test Project* (no date). Available at: https://courses.cs.vt.edu/~cs5204/fall05-gback/papers/ols2002_proceedings.pdf (Accessed: April 21, 2023).
24. Hinds, N. (no date) *Kernel Korner: The Linux Test Project, Kernel korner: The Linux Test Project*. Available at: https://dl.acm.org/doi/fullHtml/10.5555/1044989.1045001 (Accessed: April 21, 2023).
25. *Testing linux, one Syscall at a time.* (no date) *LTP - Linux Test Project*. Available at: https://linux-test-project.github.io/ (Accessed: April 21, 2023).
26. *Linux Man Pages* (no date) *Linux Man Pages Online*. Available at: https://man7.org/linux/man-pages/index.html (Accessed: April 21, 2023).
27. Leon Presser Department of Electrical Engineering *et al.* (1972) *Linkers and Loaders*, *ACM Computing Surveys*. Available at: https://dl.acm.org/doi/10.1145/356603.356605 (Accessed: April 21, 2023).
28. *Tool interface standard (TIS) portable formats specification* (no date). Available at: https://refspecs.linuxfoundation.org/elf/TIS1.1.pdf (Accessed: April 21, 2023).
29. *Installing Kraft* (2020) *Unikraft*. Unikraft. Available at: https://unikraft.org/docs/usage/install/ (Accessed: April 21, 2023).
30. Hat, R. (2012) *Position independent executables (PIE), Red Hat - We make open source technologies for the enterprise*. Available at: https://www.redhat.com/en/blog/position-independent-executables-pie (Accessed: April 21, 2023).