Proposal:
Investigation into improving performance and reliability of modern device drivers

Kyle Christie
[B00415210]

Written 16/02/22
Finalised 03/03/22

Lecturer: Mark Stansfield
Research Methods In Computing
[COMP09092]

Table of Contents

Table of Figures

1. Introduction

1.1 Abstract

The objective of this proposal is to describe the research I plan to undertake while investigating device drivers and potential means to improve performance and reliability. I begin by setting out aims and objectives, justifying why I have selected the topic and considering ethics in relation to the project. I discuss the various problems that surround drivers and past research that has been carried out to not only solve or mitigate some of these issues but give better insight into why they occur and their consequences. Following this, I detail my own research which I plan to undertake including the overall methodology, tests I plan to carry out, people that I may wish to interview and share relevant details regarding this.

1.2 Aims

- Gain an understanding of the potential issues surrounding device drivers.
- Investigate further issues within an OS which could stem from device drivers.
- Discuss the implementation of device drivers across various operating systems.
- Discuss various developments and technologies which may resolve or alleviate issues with drivers.

1.3 Objectives

- Establish whether drivers continue to act as the leading cause of error within operating systems.
- Attempt to highlight various improvements which could be made to future driver maintainability.
- Test that such improvements would continue to be viable in present-day and/or future operating systems.

1.4 Justifications

This area has been chosen due to the critical nature of device drivers and the role they serve in an computer operating system. They are fundamental in using various hardware devices and providing a multitude of extensions to Operating Systems such as file systems, network protocols, anti-virus capabilities so on *(Ball et al, 2006)*. Without device drivers, a computer would be highly limited in its ability thus they are a necessity to any operating system.

Human error, lack of advancement in the field, complex interfaces and missing expertise are among various factors which have caused device drivers to hold responsibility for causing most errors in operating systems. It also seems that advancements on the topic have fallen on deaf ears or went unnoticed as it’s claimed that Linux driver code hasn’t really evolved much since the time of Unix and the many other possible solutions that worked never found major use *(Renzelmann and Swift, 2009)*.

1.5 Ethics

During interviews I must obtain the prior legal consent of my interviewee and ensure they are aware of any and all recordings taking place. During interviews and surveys I will ensure total anonymity of all subjects. This approach to anonymity must also be applied to any businesses I may work with for the project in order to protect them and their business interests. I should also consider relevant law/legislation including copyright & licensing, the computer misuse act and the data protection act 2018.

2. Technical Review

2.1 Background

2.1.1 Lack of Research

The issue of device drivers has not seen great attention from researchers *(Padioleau et al, 2006)*. It seems that the works I've used in this proposal can attest to this as they have been published between 2001 and 2012, with the latest work (as per my own findings) published by the IEEE in 2015. Some of these works have discussed technologies such as Windows XP and Linux 2.6 which released in 2001 [10] and 2005 [9] respectively.

There seems to be little innovation as previous research has focused of system architectures, new hardware and the utilisation of type-safe languages *(Swift et al, 2003)* with little consideration or development of alternative solutions (such as subsystems within the OS). Given such information, it is clear that new research & innovation is a necessity and that the validation of previous findings will inform whether or not evolution has taken place without such research and if past issues maintain a modern presence.

2.2 Causes of Error

2.2.1 Metrics

Within an Operating System, it has been found that device drivers are typically the most likely cause of computer crashes (*Kaday & Swift, 2012)*, *(Swift et al, 2003)*. There is previous evidence of this with drivers causing 85% of crashes in Windows XP *(Ball et al, 2006)*. This is not limited to just one platform as OpenBSD was found to have a higher rate of error than the Linux kernel, within the range of 1.2 to 6 times higher *(Chou et al, 2001)*. Drivers also account for the largest volume of code within an OS. There were 5 million lines of driver code in the Linux kernel (*Kaday & Swift, 2012)* which encapsulates 70% of source code *(Padioleau et al, 2006)*. Driver code has a higher error rate than the rest of kernel code with rates of 3 to 7 times higher for certain errors, up to a factor of 10 times worse *(Chou et al, 2001)*.

From these metrics, it's easy to establish that drivers (in older or past OS kernels) were a major factor in their failure. This problem was not limited to a single kernel and, in fact, can be observed across some of the largest OS/kernel providers. The volume of code also adds to the problem with

its sheer size making it the de-facto common source of errors within or connected to the kernel as whole.

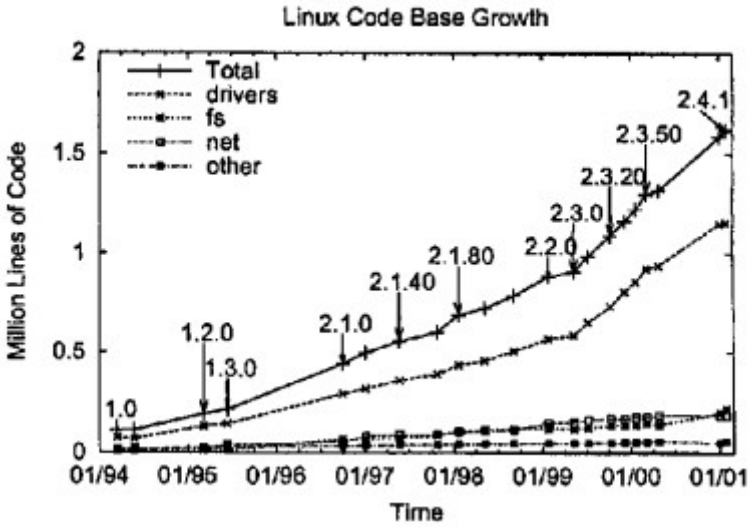

Figure 1: Chou et al. (2001) "The size of the Linux tree that we check over time"

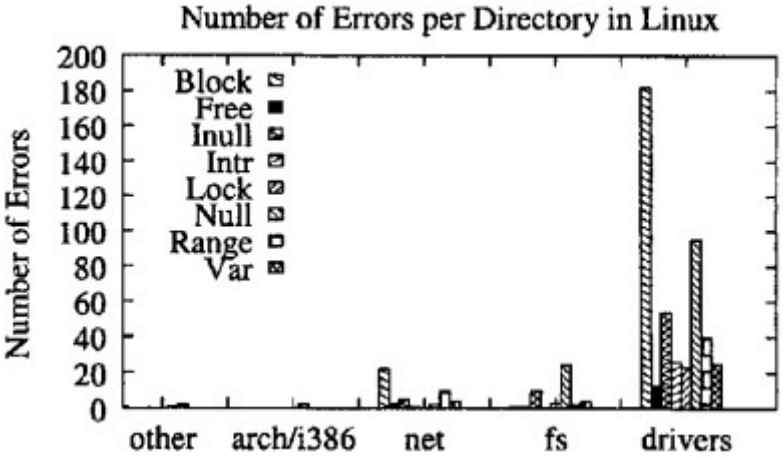

Figure 2: Chou et al. (2001) Total number of bugs found by automatic checks in the main sub-directories of Linux 2.4.1

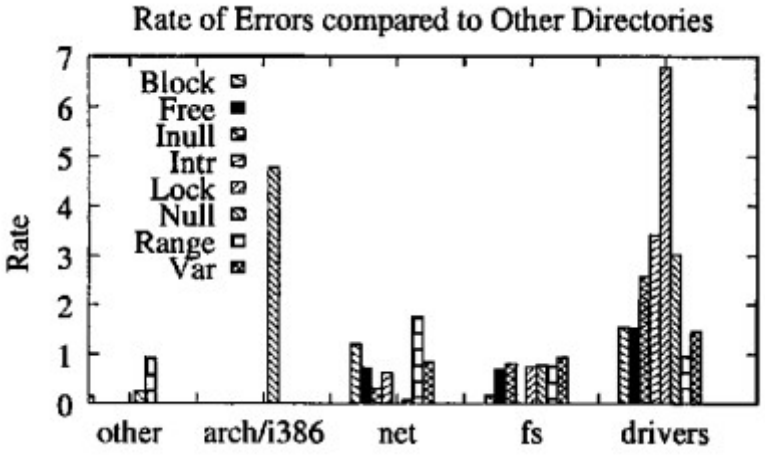

Figure 3: Chou et al. (2001) Comparison of Error rates between directories.

2.2.2 Developers

Programming device drivers is not a simple task. In the case of Windows XP, using its driver API was not easy as programmers found it difficult to use *(Ball et al, 2006)*. Window XP had many kernel APIs which were easily misused, leading to irregular behaviour or crashing *(Ball et al, 2006)*. Writing C code for the kernel also proves difficult as concluded by Renzelmann and Swift *(Renzelmann and Swift, 2009)*. It is necessary that driver developers also hold knowledge of the kernel and similar OS programming, it was found that "driver maintainers are not kernel experts but instead experts in a given device or even ordinary users who find that their hardware is not adequately supported" *(Padioleau et al, 2006)*, these developers hold little knowledge or experience in kernel organisation and programming *(Swift et al, 2003)* where "writing a robust device driver requires a great deal of expertise and precise understanding of how drivers are supposed to interact with the operating system or kernel" *(Ball et al, 2006)*. Drivers were found to be difficult to test as traditional software testing is not suitable for both the regular and more subtle errors (which occur at a low level) in drivers *(Ball et al, 2006)*.

It's clear that developers can contribute to the problem facing device drivers. Not only can they lack the necessary expertise, they may also face a complex battle with the interfaces that allow them to provide driver functionality and extensions to the kernel. Among the issues with developers, they struggled to successfully test drivers through traditional means which explains why some bugs persisted within the Linux kernel an average of 1.8 years before being fixed *(Chou et al, 2001)*.

2.2.3 Code

There are numerous issues related to driver code from little evolution to design choice and more. Possibly the largest issue facing device is drivers is the collateral evolution. Collateral evolution is

described as a consequence of evolution within the rest of the kernel where the services and functionalities used by drivers undergo change which then prompt the respective driver code to be modified to restore its previous behaviour *(Padioleau et al, 2006)*. Such modifications typically don't generate improvements to the code, collateral evolutions can account for up to 35% of modified driver code *(Padioleau et al, 2006)*.

The effort placed in resolving collateral evolutions can be complex, requiring a large amount of code reorganisation, time and expense which in turn makes the code more prone to errors *(Padioleau et al, 2006)*. In the case of Linux, collateral evolutions are made worse due to the difficulty in effectively sharing information about necessary changes to driver maintainers and developers *(Padioleau et al, 2006)*. It is believed that there are little to no tools which can assist developers in dealing with collateral evolutions. Though Muller, G et al appear to have strived to help solve this in developing 'Coccinelle' which aims to assist with collateral evolutions [8].

With links to collateral evolution, drivers also hold high dependence on the rest of the OS with support libraries and access to necessary data structures & functions provided by the kernel with 51% of driver code dedicated to "initialisation, clean up and configuration" (*Kaday & Swift, 2012)* it is no surprise that "8% of driver code is substantially similar to code elsewhere ..." (*Kaday & Swift, 2012)*. There are solutions to this which are discussed in the final section of this technical review. Device drivers continue to be written in C, similarly to how they were written in the early days of Unix *(Renzelmann and Swift, 2009)* thus, in some kernels, there has not been much change or evolution in the development of drivers.

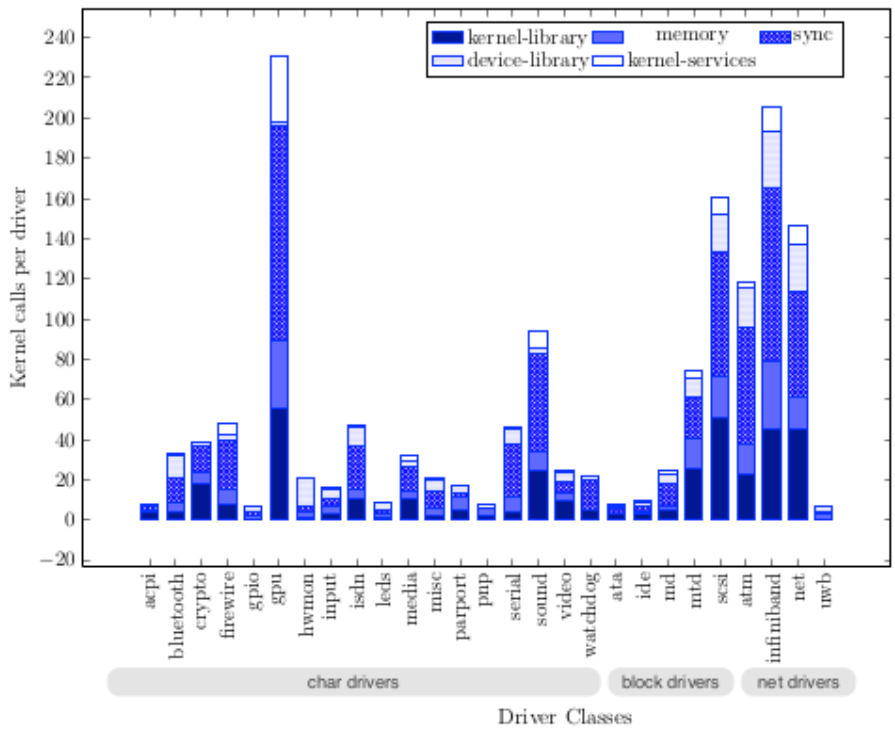

Figure 4: Kadav and Swift. (2012) An average of various library and system calls within several driver types from the Linux 2.6.37.6 kernel

2.3 Potential Solutions

2.3.1 Abstractions

The use of Object-oriented programming is said to benefit device drivers. It is believed that use of the paradigm along with inheritance and related designs can reduce the high amounts of code used in initialisation (*Kaday & Swift, 2012)*. The introduction of object-oriented programming can lead to the use of more modern languages such as C++, Java and Rust. Possibly Python and Kotlin. The use of classes would allow for the simplified implementation of API's and/or their components. This is further aided by inheritance and polymorphism which would reduce the volume of repeating code patterns and facilitate easier methods of adding to or extending API's. However, this considers design more than practical implementation. The use of C is necessary in current device drivers as it provides low-level system calls which permit communication and interaction between the driver software and the rest of the kernel. It's likely that a compromise would need to be met between retaining lower-level functionality and taking steps to introducing some object-oriented abstractions which help in reducing the volume of driver code as previously discussed.

Another proposed solution is an OS subsystem. Nooks OS is such where extensions execute in smaller, lightweight "sub" kernels *(Swift et al, 2003)* that have limited write access to the larger main kernel. Nooks also provides automatic recovery services that both track and validate changes in kernel data structures effectively preventing bugs in real time and resulting failure that could occur. It's developers believe that "improving OS reliability will therefore require systems to become highly tolerant of failures in drivers and other extensions" *(Swift et al, 2003)*. Nooks has been implemented and tested in Linux with various types of Linux kernel extensions. The authors used automatic fault injection to prove that nooks can successfully recover from errors and prevent failure. The subsystem had a low to moderate impact on driver performance. They concluded that the implementation of a real-world system like Nooks is possible and that constant evolutions in hardware will lead to an easier decision on whether such a system should be implemented and which drivers should be isolated within.

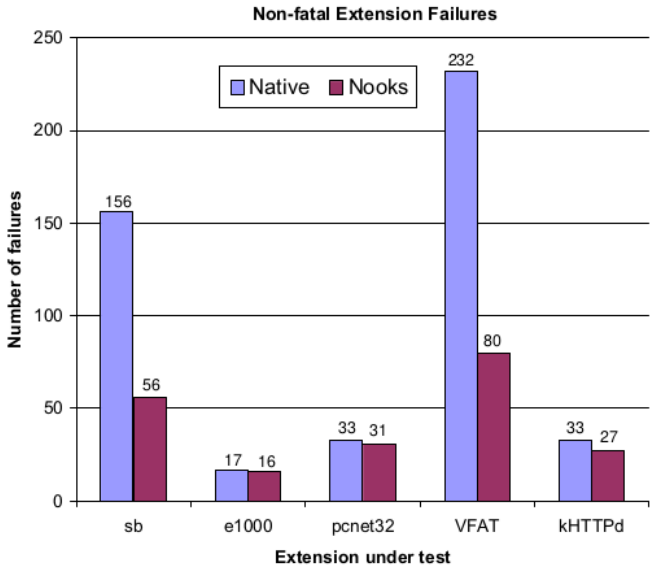

Figure 5: Swift et al. (2003) A comparison of non-fatal driver failures between Nooks and Linux kernel 2.4.18

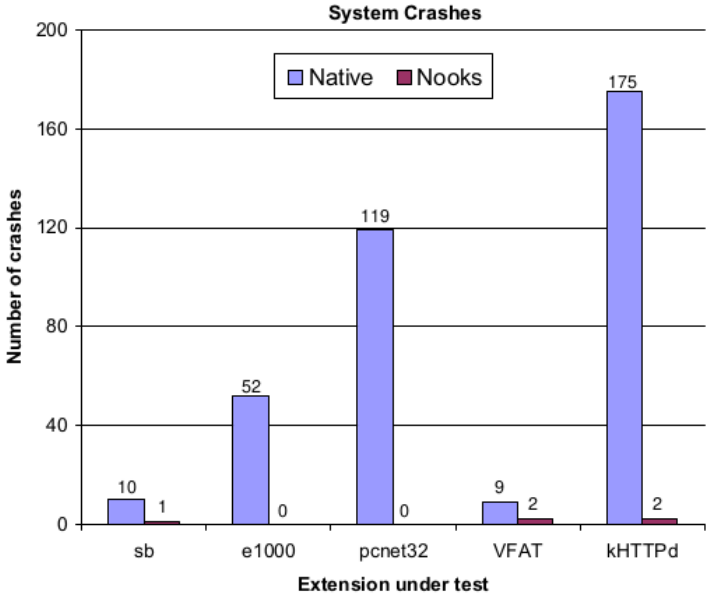

Figure 6: Swift et al. (2003) A comparison of system crashes between Nooks and Linux kernel 2.4.18.

2.3.2 Automated Tools

The Static Driver Verifier tool (developed at the Microsoft Corporation) makes use of a static analysis engine which finds instances where API rules have been broken within C programs. SDV uses this engine on device drivers to ensure proper use of relevant APIs. The tool does so by simplifying the C code into an abstract Boolean program which retains errors. In this state, API rules can be checked more efficiently and encoded as a state machine *(Ball et al, 2006)*. The Boolean program retains all Control flow instructions employing boolean variables exclusively which track the state of expressions throughout the code.

SDV automatically generates this abstraction alongside a set of expressions which should be observed. It uses a symbolic model checking algorithm based on binary decision diagrams to verify whether or not a driver obeys an API rule. To improve reliability of error reporting, the tool checks errors within the abstraction against the original C code in order to indeed verify the existence of the error. The process of abstraction, checking and refining is repeated to either provide a true error path or proof of correctness *(Ball et al, 2006)*.

It was previously found that other automatic tools were able to find errors with higher reliability and accuracy than manually checking the code *(Chou et al, 2001)*. I feel that the use of tools such as SDV, if used, will certainly aid in improving driver robustness and reliability. The tool allows for the discovery of deep/subtle errors within Windows driver thus I believe that such tools could be used more commonly in the testing of device drivers. Though authors noted that SDV is not without its own errors *(Ball et al, 2006)*.

2.3.3 Modern Languages

‘Decaf’ is a system tested on Linux and developed by Matthew Renzelmann and Michael Swift at the University of Wisconsin-Madison *(Renzelmann and Swift, 2009)*. It converts Linux kernel drivers written in C to user programs in Java. Though, performance critical code is untouched using C in the kernel. The remaining code runs in Java through a customised kernel interface. It’s believed that utilising type-safe language such as Java will improve reliability by simplifying programming, utilising exceptions (improved error handling) and permitting the use of debugging tools which previously were previously found to be unsuitable for C drivers.

The main advantages of decaf are as follows:

- Move a large amount of driver code outside of the kernel
- Reduce size of driver code
- Use exceptions to find non-functional error handling during compilation
- Evolve alongside changes within driver & kernel code and to respective data structures
- Perform within 1% of native kernel-only drivers *(Renzelmann and Swift, 2009)*

Using a system such as decaf allows for quick and easy conversion into Java, permitting a “gradual migration away from C” *(Renzelmann and Swift, 2009)*. Using Java exceptions led to various improvements such as reduced amount of code and fixing 28 cases of missing error handling. Updates to the driver required changes to Java code only. Java is not the sole language suitable for this system, the authors claim any language could be used for its implementation *(Renzelmann and Swift, 2009)*.

Clearly, a system such as Decaf, can seriously ease issues stemming from device drivers by reducing the size of several aspects, improving error handling and retaining high performance. Decaf seems to be a promising solution which can be easily employed.

3. Research Methodology

I have considered many research methodologies for use within this project. I’ve came to the conclusion that a mix of both qualitative and quantitative methods will benefit my research. Qualitative research allows for the exploration of peoples emotions, experiences or feelings and sees the use of tasks involving people such as focus groups, interviews, observations and case studies. Quantitative research focuses on statistics, facts, figures and similar numeric data derived from experiments and surveys.

In this project, a mix will be necessary as there are various presented factors that encompass both methods. It will allow me to re-test certain research tasks carried out in previous papers and gather new figures based on current, modern technology while also approaching developers to ascertain their experiences & thoughts on various relevant topics as I will later discuss.

The hope for the wider investigation is to generate present day findings and compare these to those of the past in order to clearly establish if drivers are still a major source of errors within operating systems and the potential reasons why this is or isn’t the case.

Given the accessibility of open source software, I believe that measuring the size of driver code against the rest of an OS or Kernel code base will be highly beneficial as I can record its current size and compare that to past findings. In my technical review, I noted that 70% of Linux kernel source code encapsulates drivers (*Kaday & Swift, 2012)*, I plan to verify whether this continues to be the case and if this is true within other open source operating systems or kernels. This can be easily carried out by downloading the source code from an online repository (such as GitHub or GitLab) and calculating directory and files sizes. These numbers can then be used to compare the driver source code against the rest and verify whether a large portion of total source code is indeed dedicated to device drivers. As some Operating Systems are not open source (Windows, MacOS), it is not as simple as downloading the source code because its simply not publicly available.

I also wish to focus on the use of automated tools and test these on the latest versions of major Operating Systems such as Windows 10/11, A set of Linux distributions and OpenBSD distributions. I wish to carry out the same methods used by the original researchers but with modern

technology. This will prove that such tools continue to be viable in today's operating systems. For instance, with the Nooks subsystem, the researchers used automatic fault injection on drivers running within the system. This then allowed them to prove the robustness of the system and compare performance and failure recovery in the original OS and with Nooks.

I'd also like my investigation to include driver developers by way of interviews, focus groups and/or surveys. The underlying reasons behind this are to ask about their experience, ask if they are aware of collateral evolutions, their thoughts on the current world of driver development and what they believe may be a pressing issue regarding device drivers. In my technical review, it was established that among various factors facing developers, they often lack expertise in Kernel/OS programming & organisation and are likely to know more about the device they are developing for *(Padioleau et al, 2006), (Swift et al, 2003)*. It is necessary to conduct these interviews to once again verify whether past findings remain true and to better understand the state of device drivers from the perspective of developers themselves. I especially feel that individual non-affiliated open-source driver developers may contribute to the hypothesis of developers having more knowledge on a specific device than knowledge on the necessary OS programming. Therefore, if possible, I'd like to involve both the open source community and closed source businesses to test this theory. I could also discuss my findings and if they feel they would benefit from using automatic tools or the use of Object-oriented programming within drivers. If carried out, this would very much be a open free-flowing discussion between all involved.

4. Bibliography

Kadav, A. Swift M, M.(2012) Understanding modern device drivers [Online] ACM. Available: https://dl.acm.org/doi/pdf/10.1145/2248487.2150987 [Accessed 11 February 2022]

Chou, A. Yang, J. Chelf, B. Hallen, S. Engler, D. (2001) An empirical study of operating systems errors [Online] ACM. Available: https://dl.acm.org/doi/pdf/10.1145/502034.502042 [Accessed 11 February 2022]

Padioleau, Y. Lawall L, J. Muller, G. (2006) Understanding collateral evolution in Linux device drivers [Online] ACM. Available:https://dl.acm.org/doi/abs/10.1145/1217935.1217942 [Accessed 12 February 2022]

Ball, T. Bounimova, E. Cook, B. Levin, V. Lichtenberg, J. McGarvey, C. Ondrusek, B. Rajamani K, S. Ustuner, A. (2006) Thorough static analysis of device drivers [Online] ACM. Available: https://dl.acm.org/doi/abs/10.1145/1218063.1217943 [Accessed 12 February 2022]

Deligiannis, P. Donaldson F, A. Rakamaric, Z. (2015) Fast and Precise Symbolic Analysis of Concurrency Bugs in Device Drivers [Online] IEEE. Available: https://ieeexplore.ieee.org/abstract/document/7372006 [Accessed 13 February 2022]

Renzelmann J, M. Swift M, M. (2009) Decaf: moving device drivers to a modern language [Online] University of Wisconsin-Madison. Available: https://static.usenix.org/events/usenix09/tech/full_papers/renzelmann/renzelmann.pdf [Accessed 15 February 2022]

Swift M, M. Bershad N, Brian. Levy M, H. (2003) Improving the Reliability of Commodity Operating Systems [Online] University of Washington. Available: http://nooks.cs.washington.edu/nooks-tocs.pdf [Accessed 15 February 2022]

Coccinelle (2022) Coccinelle Info – Collateral Evolutions [Online] Available: https://coccinelle.gitlabpages.inria.fr/website/ce.html [Accessed 15 February 2022]

The Linux Foundation (2012) The 2.6.32 Linux Kernel [Online] Available: https://www.linuxfoundation.org/blog/the-2-6-32-linux-kernel/ [Accessed 16 February 2022]

Wikipedia (2022) Windows XP [Online] Available: https://en.wikipedia.org/wiki/Windows_XP [Accessed 17 February 2022]

Microsoft (2021) Windows Hardware Developer: Static Driver Verifier [Online] Available: https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/static-driver-verifier [Accessed 17 February 2022]