University of Texas at El Paso
ScholarWorks@UTEP

Open Access Theses & Dissertations

2022-12-01

Analyzing And Quantifying The Impact Of Software Diversification On Return-Oriented Programming (rop) Based Exploits

David Reyes
University of Texas at El Paso

Follow this and additional works at: https://scholarworks.utep.edu/open_etd

Part of the Computer Sciences Commons

Recommended Citation

Reyes, David, "Analyzing And Quantifying The Impact Of Software Diversification On Return-Oriented Programming (rop) Based Exploits" (2022). *Open Access Theses & Dissertations*. 3718.
https://scholarworks.utep.edu/open_etd/3718

This is brought to you for free and open access by ScholarWorks@UTEP. It has been accepted for inclusion in Open Access Theses & Dissertations by an authorized administrator of ScholarWorks@UTEP. For more information, please contact lweber@utep.edu.

ANALYZING AND QUANTIFYING THE IMPACT OF SOFTWARE DIVERSIFICATION ON RETURN-ORIENTED PROGRAMMING (ROP) BASED EXPLOITS

DAVID REYES

Doctoral Program in Computer Science

APPROVED:

Salamah Salamah, Ph.D., Chair

Jaime Acosta, Ph.D.

Deepak Tosh, Ph.D.

Sai Mounika Errapotu Ph.D.

Stephen L. Crites, Jr., Ph.D.
Dean of the Graduate School

Copyright ©

by

David Reyes

2022

DEDICATION

To my mother, father, sister, nieces, and fiancé.

Your love, support, and patience have made all this possible. It has meant the world to me.

Finally, to my grandfather, Lorenzo Reyes Sr., who sadly is not here to witness this.

Ahora si, ya acabe Apa.

ANALYZING AND QUANTIFYING THE IMPACT OF SOFTWARE DIVERSIFICATION ON RETURN-ORIENTED (ROP) BASED EXPLOITS

by

DAVID REYES, BSCS, MSSwE

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at El Paso

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

Department of Computer Science

THE UNIVERSITY OF TEXAS AT EL PASO

December 2022

ACKNOWLEDGMENTS

I want to thank first and foremost, my committee Dr. Salamah Salamah, Dr. Jaime Acosta, Dr. Deepak Tosh, and Dr. Sai Mounika Errapotu. Thank you all for your time and for participating in this journey.

To Dr. Salamah Salamah and Dr. Jaime Acosta, thank you both for your support, motivation, guidance, and most importantly, **patience,** without which this work would not have been possible. Thank you both for being role models, mentors, teachers, and guides. Thank you, Dr. Salamah Salamah, for constantly pushing me to excel. From the moment I first stepped into your Software II class, throughout the Master's program, and finally during the Ph.D. process, I know you have aged as much as I have. Thank you for entrusting me with programs such as the Cyber Patriot Initiative and for putting more than I thought I could handle on my plate. But more so, thank you for pushing me to keep going when I was ready to quit, believing in me when I didn't, and helping me through the lows.

I also would like to thank Dr. Ann Q. Gates for your mentorship during my time as a student in your Software Engineering I course and as your Teaching Assistant. I have learned so much about software engineering, teaching, and mentorship from you, and I will take it with me in this next chapter of my career.

I also want to thank the friends I made at UTEP, Pedro, Daniel, and Pruitt. Thank you, guys, for being some of the most incredible friends in and out of UTEP. I have learned a lot from the three of you, and I hope to continue to learn from you all. I'll see you all at Daniel's lake house.

I would also like to thank the entire Computer Science department faculty and staff for all of the work that they do. While I will miss the department while I am away, I plan to return

eventually and teach what I learned from my time in the real world to the next generation of students.

Finally and certainly not least, I want to thank a significant person in my life, my fiancé, Cynthia Morales-Contreras; thank you for listening when I needed to vent, providing me with someone to talk to during the hard times, and for always offering feedback over these past years.

ABSTRACT

With the implementation of modern software mitigation techniques such: as Address Space Layout Randomization (ASLR), stack canaries, and the No-Execute bit (N.X.), attackers can no longer achieve arbitrary code execution simply by injecting shellcode into a vulnerable buffer and redirecting execution to this vulnerable buffer. Instead, attackers have pivoted to Return Oriented Programming (ROP) to achieve the same arbitrary code execution. Using this attack method, attackers string together ROP gadgets, assembly code snippets found in the target binary, to form what are known as ROP Chains. Using these ROP Chains, attackers can achieve the same malicious behavior as previous code injection attacks on vulnerable buffers. Furthermore, because of the static location of these ROP gadgets, attackers can re-use their exploit code across all systems running the binary. This phenomenon is what is called a write-once, compromise-everywhere scenario.

Software diversification has been presented as a possible mitigation strategy over the past seventeen years. Software diversification is a technique that modifies the instructions in binaries while maintaining their semantic behavior. The means given the same input binaries would produce the same output; however, the diversified binary is syntactically different at the assembly level.

Previous work in this area has shown general success in reducing the number of shared gadgets. However, there has been a lack of research that analyzes how diversification affects an attacker from re-using a previously crafted exploit. Furthermore, current research has not presented approaches that measure diversification algorithms' impact and effectiveness on binaries. Finally, because software diversification modifies the assembly code of binaries, different binaries are affected in vastly different ways. In addition to the different diversification algorithms, defenders

can find it challenging to determine which configurations best suit their needs. This uncertainty may lead to unwanted trade-offs; for example, one diversification algorithm might make it harder for modern tools like Fuzzers to find crashes or vulnerabilities. The impact might come at the cost of increasing the total number of gadgets in the binary or increasing the program's run time. Likewise, while one algorithm might offer protection while minimizing the number of ROP gadgets, it might allow modern tools or attackers to locate the vulnerability faster than if another algorithm were applied.

To address the lack of research in this area, the work presented in this dissertation analyzes software diversification's impact on exploit re-use attacks, identifies the primary criteria to quantify the efficacy of diversification algorithms, and proposes a method to quantify the effectiveness of diversification algorithms. Finally, this work develops and presents a system that identifies the appropriate algorithm(s) or combination of algorithms based on the end user's needs using the quantification methods developed. This system allows the end user to quickly and easily identify the appropriate algorithm based on their security preferences or requirements; while giving the end user an understanding of the trade-offs between algorithms. With this understanding, the end user can create multiple diversified variants of the target binary that meet their security needs.

TABLE OF CONTENTS

DEDICATION III

ACKNOWLEDGMENTS V

ABSTRACT VII

TABLE OF CONTENTS IX

LIST OF TABLES XIII

LIST OF FIGURES XIV

CHAPTER 1: INTRODUCTION 1

1.1 MOTIVATION 1

1.2 RESEARCH PROBLEM 5

1.3 SIGNIFICANCE OF THE RESEARCH 6

1.4 ORGANIZATION OF THE DISSERTATION 8

CHAPTER 2: BACKGROUND 9

2.1 EXPLOITATION TECHNIQUE: BUFFER OVERFLOW 9

2.2 EXPLOITATION TECHNIQUE: RETURN-ORIENTED PROGRAMMING (ROP) 12

2.3 DIVERSIFICATION TECHNIQUES 16

2.3.1 Implementation Time 16

2.3.2 Compiling and Linking Time 17

2.3.3 Installation Time 20

2.3.4 Load Time 25

2.3.5 IoT Devices 28

2.4 SUMMARY 29

CHAPTER 3: RELATED WORK 31

3.1 MEASURING THE EFFECTIVENESS OF DIVERSIFICATION APPROACHES 31

3.1.1 Diversification on Gadget Removal 31

3.1.1 Diversification on Binary Performance 34

3.2 SUMMARY 38

CHAPTER 4: RESEARCH QUESTIONS 39
4.1 RESEARCH GOAL....39
CHAPTER 5: METHODOLOGY 41
5.1 EXAMINING THE IMPACT OF SOFTWARE DIVERSIFICATION ON EXPLOIT DEVELOPMENT 41
5.1.1 Experimental Overview 41
5.1.1.1 Diversification Engine Selection 41
5.1.1.2 Binary Selection....43
5.1.1.3 ROP Gadget Analysis Toolset 45
5.1.1.4 Experimental Design....46
5.2 QUANTIFYING SOFTWARE DIVERSIFICATION 47
5.2.1 Identifying the Appropriate Quantification Metrics 48
5.2.2 Quantifying Attack Resistance 48
5.2.3 Quantifying Exploit Complexity....50
5.2.4 Quantifying Resistance to Reverse Engineering....51
5.3 IMPLEMENTATION OF THE SELECTOR SYSTEM FOR DIVERSIFIED BINARIES 52
5.3.1 Choice of Implementation Platform....53
5.3.2 System Components....54
5.3.3 High-Level System Design 54
5.3.4 Execution Component....55
5.3.5 Diversification Component....56
5.3.6 Analysis Component....57
5.3.7 Algorithm Selection Component 58
5.3.8 Visualization Component....59
5.4 SUMMARY 60
CHAPTER 6: RESULTS AND OBSERVATIONS....61
6.1 IMPACT OF SOFTWARE DIVERSIFICATION ON EXPLOIT DEVELOPMENT ..61
6.1.2 Gadget Count 61
6.1.2 Surviving Gadgets....64
6.1.3 Exploit Generation 68
6.2 QUANTIFICATION OF THE IMPACT OF SOFTWARE DIVERSIFICATION 72
6.2.1 Diversification on Attack Resistance....72

6.2.2 Diversification on Exploit Complexity 74
6.2.3 Diversification on Resistance to Reverse Engineering 75
6.3 ALGORITHM SELECTION CASE STUDY 76
6.3.1 Setup 76
6.3.2 Maximizing Attack Resistance 77
6.3.3 Balanced Diversification 81
6.4 DISCUSSION 85
CHAPTER 7: CONCLUSION 87
7.1 SUMMARY 87
7.2 FUTURE WORK 88
REFERENCES 91
GLOSSARY 98
Definitions 98
Acronyms 98
APPENDIX A 99
Examples of ROP Gadgets 99
APPENDIX B 100
Examples of System Calls 100
APPENDIX C 101
APPENDIX D 102
APPENDIX E 103
APPENDIX F 104
APPENDIX G 105
APPENDIX H 106
APPENDIX I 107
Complete Attack Resistance Results 107
APPENDIX J 115
Complete Exploit Complexity Score Results 115

APPENDIX K 122
Complete Resistance To Reverse Engineering Score Results 122

VITA 129

LIST OF TABLES

Table 1: Average Percentage of Shared Gadgets between Variants and Non-Diversified Binaries 65
Table 2: Attack Resistance Score Results 72
Table 3: Exploit Complexity Score Results 74
Table 4: Resistance to Reverse Engineering Score Results 75
Table 5: Definition of Terms Used 98
Table 6: Acronyms 98
Table 7: ROP Gadgets 99
Table 8: Exit System Call Values 100
Table 9: Attack Resistance Score Results for All Binaries 107
Table 10: Exploit Complexity Score Results for All Binaries 115
Table 11: Resistance to Reverse Engineering Score Results for All Binaries 122

LIST OF FIGURES

Figure 1: One Version Software 3
Figure 2: Software Diversification Approach 4
Figure 3: Buffer Overflow Example 11
Figure 4: ROP Chain Example 13
Figure 5: Full *execve* ROP Chain 15
Figure 6:Unaligned *syscall* Gadget Found by ROPPER 46
Equation 1: Attack Resistance Formula 49
Figure 7: Example of a ROP gadget that clobbers registers 50
Figure 8: Selection System Architecture 55
Equation 2: Normalization Formula 58
Equation 3: Impact Score Formula 59
Figure 9: Function and Gadget Relationship Non-Diversified: Real-World 62
Figure 10: Function and Gadget Relationship All Algorithms and Non-Diversified: Real-World 63
Figure 11: Function and Gadget Relationship All Algorithms and Non-Diversified: Coreutils .. 64
Figure 12: Surviving Gadgets between Original and Variants PDFResserect 67
Figure 13: ROPPER Generated execve for Sipp 68
Figure 14: ROPPER Generated mprotect Exploit for Crossfire-Server 70
Figure 15: ROPPER Generated mprotect Exploit for Crossfire-Server All Diversification Algorithms Applied 70
Figure 16: Base64 Best Attack Resistance Algorithm 78
Figure 17: CPU-Time Impact for Base64 79
Figure 18: Sha512Sum Best Attack Resistance Focus 80
Figure 19: CPU-Time Impact for Sha512 Sum 81
Figure 20: Base64 Best Balanced 82
Figure 21: Base64 Total Execution Time 83
Figure 22: ls Best Balanced 84
Figure 23: ls Total Execution Time 85
Figure 24: Shred CPU Clock Cycles Plot 101
Figure 25: Shred ROP Gadgets Plot 102
Figure 26: CP CPU Clock Cycles Plot 103
Figure 27: CP ROP Gadgets Plot 104
Figure 28: ls CPU Clock Cycles Plot 105
Figure 29: ls ROP Gadgets Plot 106

CHAPTER 1: INTRODUCTION

Security vulnerabilities exist in many domains, including networking, operating system, and application. Security risks at the application level are some of the most significant security problems impacting systems today. These applications are connected to the cloud and are now often available over various other networks. Failure to address security throughout the application lifecycle can result in catastrophic damages like the loss of intellectual property, money, or data [1]. A large number of critical systems further exacerbates this point. These systems contain features such as low-level support, optimizations, and interfacing with hardware components. However, the responsibility of securing these systems is almost exclusively to the programmer. Coding issues such as user input sanitization, input bounds checking, or managing dynamically allocated memory correctly can introduce vulnerabilities that might not be found in testing. This code composes both small and large systems alike and is used across enterprises, with their vulnerabilities unknown. [2].

1.1 MOTIVATION

Significant effort has been made to secure systems over the past fifteen years to address the risk of coding issues. With the implementation of mitigation techniques such as stack canaries [3], ASLR [4], and DEP [4], there has been an increase in the security of computer networks and operating systems. However, the dynamic nature of securing software systems means attackers continue successfully developing new methods and combining already-established ones to achieve malicious actions. One method attackers have begun using to get around security mitigations is an approach called Return Oriented Programming (ROP), also known as code re-use attack. Under the right circumstances, ROP can allow attackers to execute arbitrary code on the vulnerable

program. This exploit is achieved through assembly code snippets called 'gadgets' which can be linked together to execute larger commands. The linking of more than one ROP gadget is called an ROP chain. One example of using this method occured in November of 2018 when a security researcher using the pseudonym MorteNoir1 identified a zero-day vulnerability. This vulnerability is unknown to the developers; therefore, it has been zero days since it was patched and created an exploit with the ability to escape a virtualized environment. After escaping this virtualized environment, the exploit could run arbitrary code on the host machine. This exploit also had advanced capabilities that bypassed modern defenses such as ASLR and DEP. The researcher then used a stack and heap overflow to gain control of the program's execution flow. Finally, using a series of ROP gadgets, the researcher created a ROP chain to escape the virtual machine and cause arbitrary code execution. This arbitrary code execution allowed him to escape from the guest VM to the host system [5]. This example is only one of the many situations where attackers have used ROP to gain arbitrary code execution in a program. We can expect these attacks to continue without a way to secure a software program.

Furthermore, a single binary representation is distributed and installed when software is distributed to numerous systems. The consequence of having identical binaries is that a security vulnerability exploiting a particular binary will make all environments where the software system is installed susceptible to the same exploit. Consequently, an attacker only has to develop a single exploit that can impact a wide range of users. Figure 1, initially presented in [6], illustrates this point.

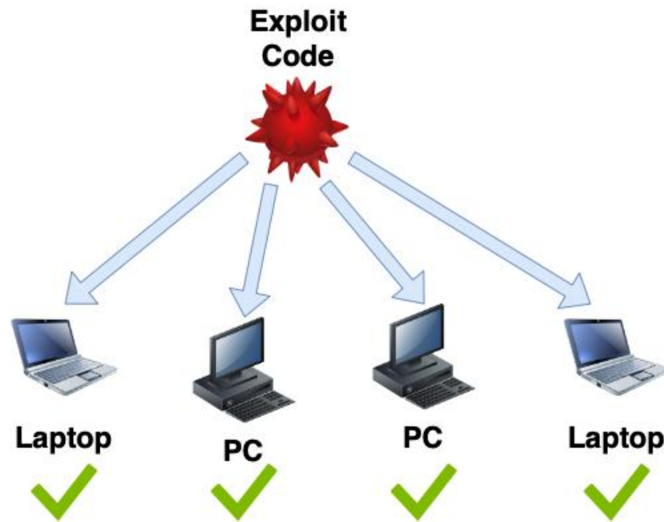

Figure 1: One Version Software

Software diversification is an approach to software defense that creates unique variants of the target program, given either source code or binary code. These variants are semantically equivalent but have syntactically different assembly codes. This approach's advantage is the availability of different binaries for the same system, which means that other users will install variant binaries in their particular environment. This means that if an attacker develops or finds an exploit for one of these variants, it is not guaranteed that the exploit will work as the assembly code would differ on each variant. Thus critical pieces for the exploit could be missing or located in different offsets than expected. This approach to software protection, which is not widely adopted in software development, adds a layer of uncertainty to the target program. Knowledge obtained by the attacker from one binary would then not apply to other copies. This approach can be shown in Figure 2.

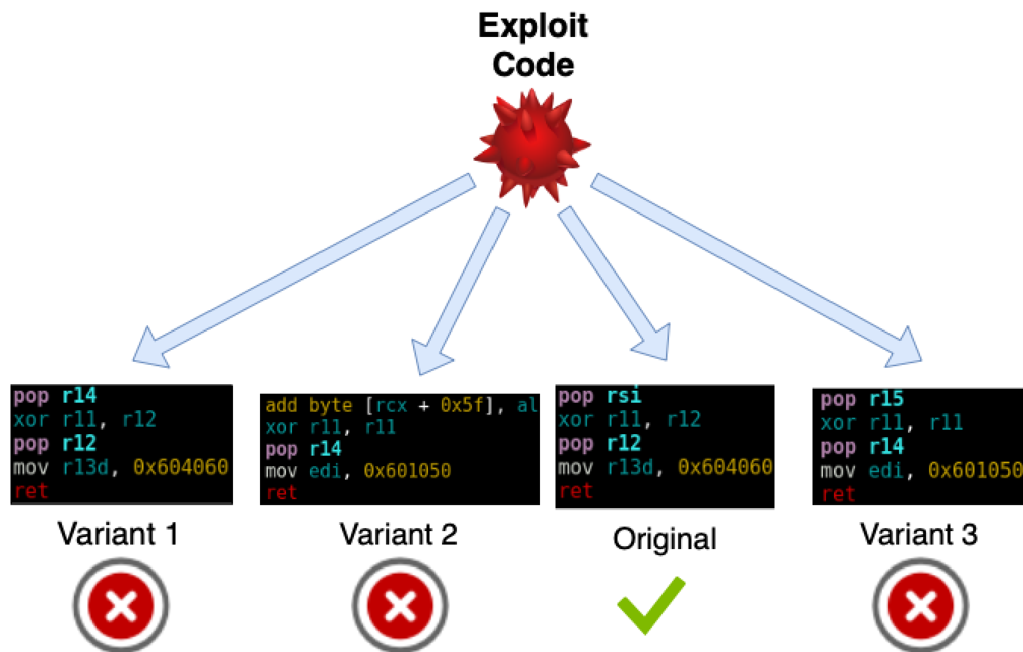

Figure 2: Software Diversification Approach

As seen in the figure, the assembly code for each variant is different. These subtitle differences in assembly code would be enough to stop an attacker's code from being executed, as each would produce unexpected or unwanted behavior. However, as noted in [7], most current software diversification approaches have remained primarily academic. It is not widely known whether software diversification will add any benefit to compiled binaries. This work aims to help understand if diversification offers benefits when applied to binaries and if software diversification would make it more difficult for an attacker to use pre-written exploits. There are many possible reasons for software diversification's lack of mainstream acceptance. One reason has been presented in [8]. In this work, the authors identify that there is currently a lack of research that quantifies the impact diversification introduces from an exploitability standpoint. Therefore it is not entirely known if diversification helps. Previous work in diversification has been heavily

focused on approaches and where and how diversification can be applied, i.e., upon compilation, through binary re-writing.

Additionally, while most previous works have reported the impacts their diversification engines have on performance, very few report on eliminating ROP gadgets and machine instructions are already present in the code. Those that do report on the impact diversification has on ROP gadgets use automated tools like ROPGadget [9], Mona [10], and Q [11] to try to find gadgets. The use of these tools is an issue because most are relatively sensitive to change and approaches, such as inserting a no-operation instruction (NOP). NOP instructions do not modify the program's processing state and can fool these tools into thinking gadgets are eliminated. In some works, gadget elimination is determined by whether a gadget still exists in the same memory offsets across variants. Most of these tools used to find gadgets are not the only tactics used by an attacker. Furthermore, skilled attackers might catch on that diversification tactics are being utilized and adjust their exploits accordingly.

1.2 RESEARCH PROBLEM

Modern exploitation techniques leverage a mechanism called return-oriented programming (ROP). ROP gadgets are defined as: a sequence of short meaningful instructions that are part of an executable. Attackers can chain together ROP gadgets to create a ROP chain. When these ROP gadgets are executed in a specific order they can complete a malicious action that has the same effect as injecting shellcode into a program.

For example, most modern IP security cameras' firmware comes with a built-in web service that allows the camera owner to modify the file configuration, view recordings, etc. These web services often read user input from a web form; however, an attacker could craft an exploit if the

input is not correctly sanitized or validated. These exploits can redirect control of the binary to execute attacker-requested commands (such as deleting recorded videos, viewing videos, stealing passwords, resetting passwords, etc.). While this example illustrates one scenario using an Internet of Things (IoT) device, these attacks are not limited to IoT devices or even the specific architecture associated with IoT devices, as code injection continues to be a part of Open Web Application Security Project's (OWASP) top ten vulnerabilities.

This attack is not new as it has been around for over 17 years. Since then, software diversification has been presented as a critical mechanism to thwart these types of attacks. Software diversification attempts to modify an executable so that these gadgets needed to create an exploit are no longer present, making the job harder for the attacker and adding some uncertainty from executable to executable. However, the problem is that diversification's impact on protecting against ROP exploits has not been evaluated in the research; currently, there is no actual methodology to assess the effects of different diversification techniques.

1.3 SIGNIFICANCE OF THE RESEARCH

The study of the efficacy of diversification and how it affects an adversary's effort is described by Larson et al. as an area that "is very much so in its infancy." As previously noted, [12] echoed this statement by stating, "Few studies consider how diversity interfered with exploit re-use attacks." This work takes forward steps in understanding how diversification interferes with ROP attacks and learning more about how diversification can help defend against ROP exploits. Additionally, this work presents ways to quantify the impact of software diversification, and through the development of a selector system offers a method for defenders to be able to select the best diversification algorithm for their scenarios.

The significance of this work is threefold. First, by understanding the effects of diversification, it can be understood if diversification does add extra barriers to an attacker trying to compromise diversified software systems and identify if diversification, at its current state, is a suitable way to defend against ROP exploits. Second, such an effort to develop metrics to analyze a methodology to quantify software diversification algorithms has never been attempted. Through the metrics and evaluation techniques developed in this work, researchers will also be able to compare the effectiveness of different algorithms. These techniques will also identify if combining diversification algorithms offsets the benefits of using a single algorithm in terms of performance and binary hardening. Finally, by using the quantifiable metrics developed in this work, operators can better identify diversification algorithms that best meet their needs while minimizing or understanding the trade-offs associated with diversification, not only from a binary hardening standpoint but also from a performance standpoint. With the development of these quantification models, the vision will be that this will allow researchers to have a way to measure software diversification's impact. These models will serve as a starting point, and researchers will be able to expand on them in future research.

The results from this dissertation will allow for several avenues of future research. This work will make the quantification of data more accessible, enabling researchers to develop more robust diversification engines, which will assist in efficiently eliminating or breaking up gadgets. Additionally, by extending the automated framework and tools created as part of this work, future researchers will be able to focus more attention on developing diversification algorithms and use this work to analyze those algorithms. Finally, with the toolset developed as a part of this work, operators will have access to a suite of tools to assist them in determining the best diversification algorithm(s). The suite of tools created can help operators select diversification

algorithms that best suit their needs and visualize variables that apply to them other than just ROP gadget elimination (i.e., file size, CPU cycles executed, power usage, etc.). This work will help transition software diversification into a defense method widely accepted in the general software development community.

1.4 ORGANIZATION OF THE DISSERTATION

This dissertation is organized as follows. Chapter 2 provides background on the buffer overflow vulnerability, discusses return-oriented programming (ROP) as an exploitation technique, and discusses the background work on existing software diversification approaches. Chapter 3 discusses the related work concerning software diversification and previous work on analyzing the impact of diversification approaches. Chapter 4 presents the research goals of this dissertation. Chapter 5 details the methodology regarding the analysis of software diversification's implications for exploit development, the identification of the criteria and methods proposed to quantify the impact of software diversification, and the implementation details of the selection system. Chapter 6 presents the results and observations from this work. Finally, Chapter 7 summarizes the work in this dissertation and discusses future directions in this area, followed by a glossary of terms, appendices, and references.

CHAPTER 2: BACKGROUND

Before understanding and developing methods that can assist in developing new software diversification techniques, it is first essential to understand what security vulnerabilities are and how they occur. Additionally, it is crucial to know how attackers can use these vulnerabilities to develop Return Oriented Programming (ROP) exploits that are Turing Complete [13]. Because ROP-based exploits are the successor of Buffer overflow vulnerabilities, this chapter will begin by giving a brief background on Buffer Overflow vulnerabilities, followed by a detailed explanation of what ROP is, how it works, and how an attacker can use ROP gadgets to divert a program from its normal execution. Finally, this chapter concludes with previous work in the software diversification literature.

2.1 EXPLOITATION TECHNIQUE: BUFFER OVERFLOW

Due to the Von Neumann Architecture, code and data are treated the same [14]. This lack of separation between the two allows for user data that can be executed like code to be executed, thus allowing an attacker to divert a program's execution from normal execution. The buffer overflow vulnerability was first published in 1996 by Aleph One in the e-zine Phrack and is a type of memory corruption vulnerability where more data is written to a buffer than allocated space, thus overwriting data on adjacent memory addresses [13]. This vulnerability is typically associated with programming languages such as C and C++, which hand over memory allocation and bounds checking to the programmer. This lack of bounds checking can be due to the programmer using unsafe functions such as *gets()*, which keeps reading input until it receives a newline encountered, or improper use of safe functions like *fget()*. Moreover, the memory overwriting associated with buffer overflows allows attackers to write into areas that hold executable code or overwrite a

program's state. Allowing attackers the ability to execute a set of instructions injected and diverge execution to this malicious code, historically the malicious code injected is known as shellcode.

While this attack is more than twenty years old, unfortunately it is still a relevant attack method today; from 2016 to 2022, the National Vulnerability Database documented 5,856 buffer overflow vulnerabilities in software systems [15]. Moreover, at the time of this dissertation's publication, CVE-2022-3786 was the most recent buffer overflow vulnerability. Additionally, those numbers do not consider the potential buffer overflow vulnerabilities between systems that share source code. For example, [16] found that 62% of code source code was shared between proprietary automobile firmware and open-sourced router firmware.

What makes buffer overflow vulnerabilities so dangerous is that in a traditional stack buffer overflow, attackers redirect program execution by placing malicious shellcode directly onto a vulnerable buffer and begin corrupting the adjacent memory. This allows the attacker to overwrite the return address, the next instruction executed after a function terminates execution, with the address of their shellcode. This redirection will cause the malicious shellcode to run when the program tries to return. Figure 3 details an example of this technique.

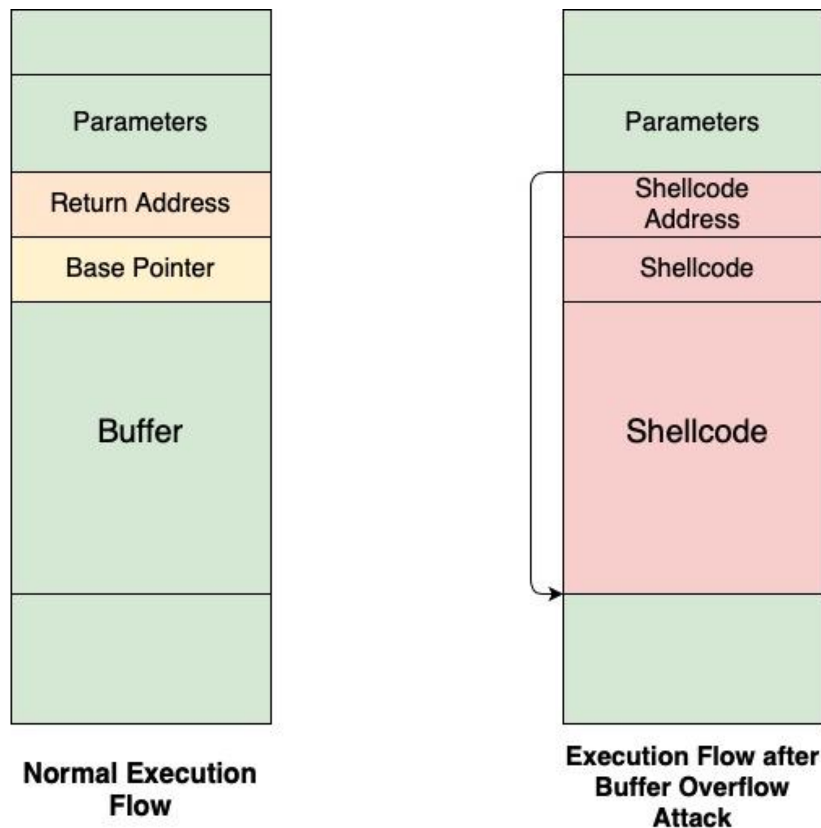

Figure 3: Buffer Overflow Example

Under normal execution flow, the return address will redirect to the next instruction to be executed after the function terminates, as shown on the left in Figure 3. As shown on the right, an attacker can use the vulnerability to place their shellcode onto the buffer, overwrite the return address, and point the return address to the shellcode on the stack. Thus when the function returns, execution would be diverted to execute this malicious shellcode. In buffer overflow vulnerabilities, the attacker is not limited to only executing shellcode. As long as the attacker has control of the return address, they can re-route execution to any location in the memory they want.

Computer architecture designers began developing mitigation techniques to treat code and data as separate entities to address the underlining problem presented by buffer overflow vulnerabilities. These mitigation strategies led to the development and introduction of Data Execution Prevention (DEP) in Windows systems and its Linux counterpart, the No-Execute bit

(NX). With the introduction of DEP in Windows systems and the No-Execute bit (NX) in Linux systems, attackers no longer have the ability to execute shellcode directly from a buffer. To get around these mitigation, attacks have shifted to a new technique called Return Oriented Programming (ROP).

2.2 EXPLOITATION TECHNIQUE: RETURN-ORIENTED PROGRAMMING (ROP)

Due to modern mitigations like Data Execution Prevention (DEP) for Windows and Non-Executable bit (NX) for Linux systems, attackers can no longer take advantage of a vulnerable buffer by injecting shellcode and redirecting execution to the address of that shellcode. As a result, modern exploit development methods used by attackers rely on an approach known as Return Oriented Programming (ROP), an exploitation tactic first presented in [17]; this exploit method bypasses mitigations like DEP/NX and achieves arbitrary code execution.

By linking, short code sequences already present in the program the attacker can achieve code execution that is Turing-Complete [18]. These code snippets are comprised of instruction sequences or immediate data words ending with a *'ret'* and have traditionally been referred to as ROP gadgets. These gadgets allow attackers to: modify registers, write/read to/from memory, and execute system calls. To create a meaningful exploit, attackers combine multiple gadgets to develop a ROP chain, a collection of one or more ROP gadgets. These ROP chains traditionally end in a system call, although that might not always be the case. Once attackers have an entire ROP chain, they can then use these ROP chains to create Turing-Complete [19] exploits that mimic the same behavior as shellcode without injecting it into the program. Because these gadgets are primarily in the *.text* section of the binary, the executable bit is enabled, allowing the attacker to

execute this malicious payload without being affected by the NX mitigation. Moreover, attackers can even disable mitigations like NX with a small set of gadgets.

A brief example of gadgets that can be used to accomplish these actions is shown in Table 7. Attackers can use ROP gadgets to create fake stack frames and set up the stack to make function calls with arguments similar to how functions would be called during normal execution. A example of how an attacker uses these ROP gadgets to execute the *exit(0)* system call is shown in

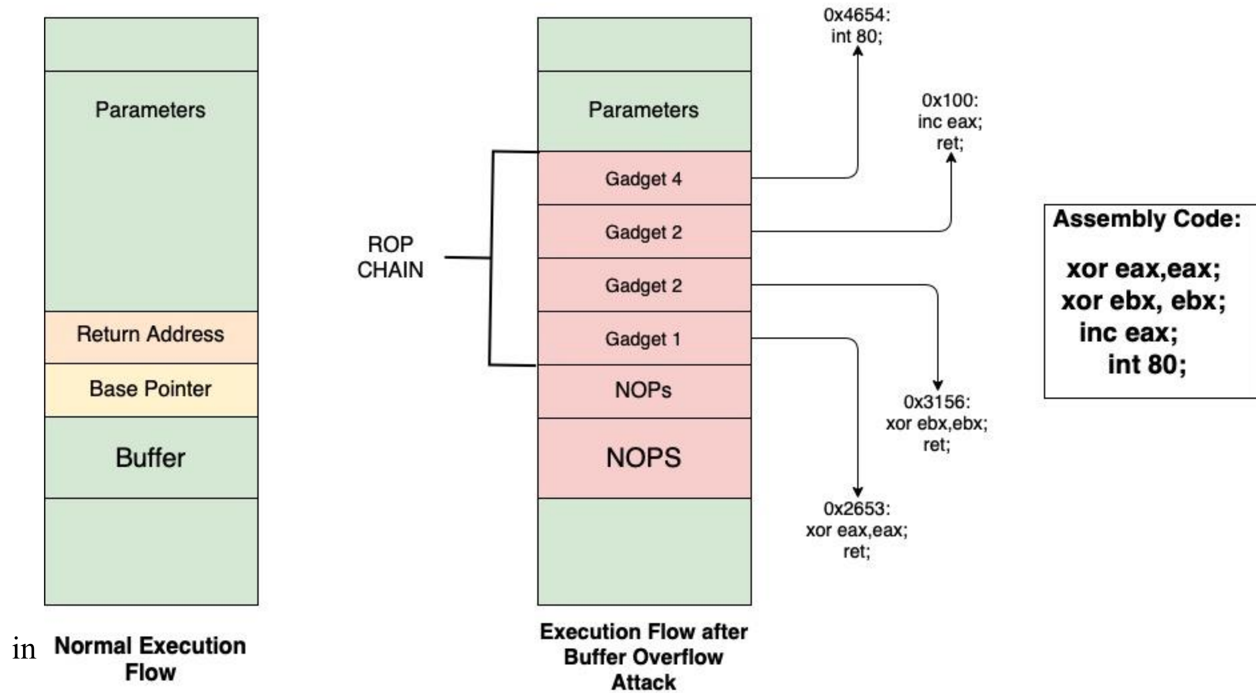

Figure 4.

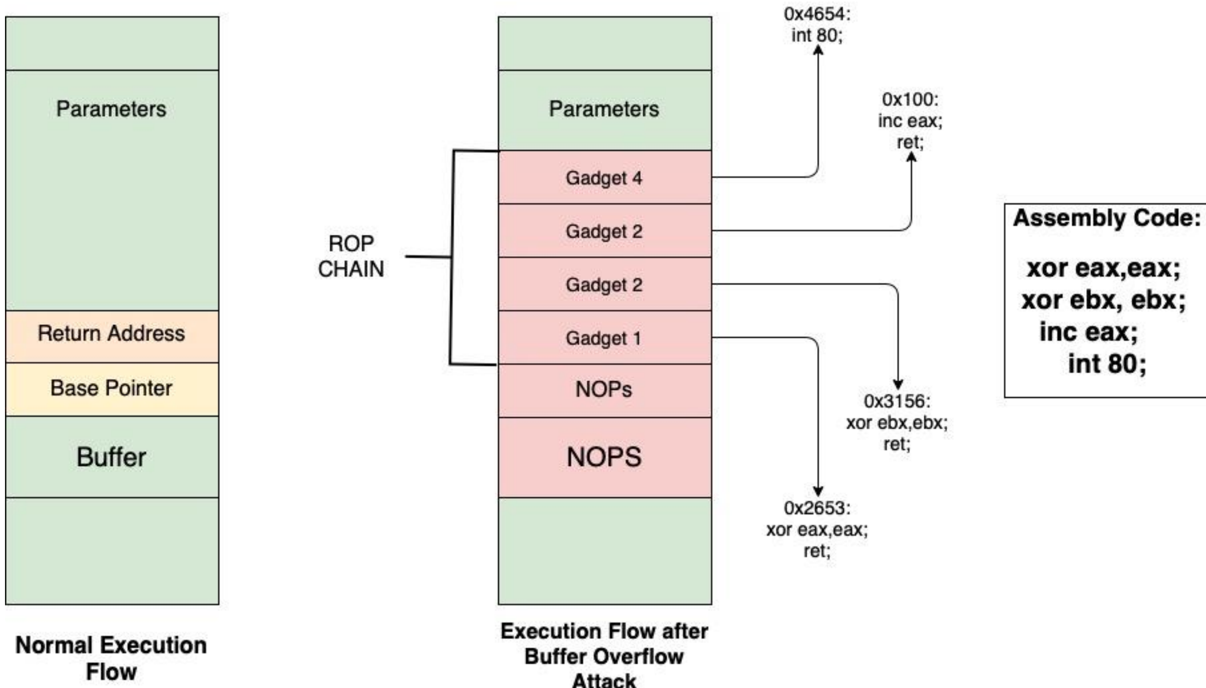

Figure 4: ROP Chain Example

From the example above, suppose that the attacker has identified a buffer overflow vulnerability and can overwrite the return address. The attacker would then look for gadgets that will allow them to execute system commands; in the example shown above, the attacker would want to run the assembly code shown that executes the system call *exit(0)* (the function call that successfully terminates the program). After identifying the vulnerabilities, the attacker would need to know the values in the registers. Table 8 details the values registers should contain for standard function calls. Using Table 8, the attacker knows that register *eax* should have a value of one (0x1), and the register *ebp* should contain the value of the exit code in this example, which would be zero. Next, the attacker would look for gadgets that allow them to meet the register value requirements. These addresses would then be placed on the stack to execute. In this example, we can see that when the function returns, instead of the original return address, the attacker will redirect execution to the address *0x2653*; this redirection will execute the instruction *xor eax, eax; ret.* When the program runs the *ret* instruction, the program will perform the second gadget, and this process will

continue until *int 80;* executes. In x86 assembly code, *int 0x80* is used to invoke a system call. While this is a simple example, more advanced exploitations do not deviate much from this example as they still rely on small gadgets to create more complex actions. The only difference is that complex ROP chains require more ROP gadgets to populate registers and read or write to or from memory, depending on the attacker's overall goal. Figure 5**Error! Reference source not found.** shows an example of a complete ROP chain. An interactive shell is executed in this ROP chain.

```python
'''
Here is the assembly equivalent for these blocks
write "/bin/sh" to 0x6b6000
pop rdx, 0x2f62696e2f736800
pop rax, 0x6b6000
mov qword ptr [rax], rdx
'''
rop = ''
rop += popRdx
rop += "/bin/sh\x00" # The string "/bin/sh" in hex with a null byte at the end
rop += popRax
rop += p64(0x6b6000)
rop += writeGadget
'''
Prep the four registers with their arguments, and make the syscall
pop rax, 0x3b
pop rdi, 0x6b6000
pop rsi, 0x0
pop rdx, 0x0
syscall
'''
rop += popRax
rop += p64(0x3b)
rop += popRdi
rop += p64(0x6b6000)
rop += popRsi
rop += p64(0)
rop += popRdx
rop += p64(0)
rop += syscall

# Add the padding to the saved return address
payload = "0"*0x408 + rop
# Send the payload, drop to an interactive shell to use our new shell
target.sendline(payload)
target.interactive()
```

Figure 5: Full *execve* ROP Chain

2.3 DIVERSIFICATION TECHNIQUES

The idea of software diversification as a way of software defense is a concept that is at least two decades old [8]. Previous work has determined that diversification can be added to a program at the design and implementation phase of the software development lifecycle and at the deployment and patching phases [20]. However, in defending against code reuse attacks, when we decide to diversify would determine the toolset and approach used. These decisions will not only affect how effective the removal of ROP gadgets is but also affect performance, CPU usage, and the resulting binary.

This section discusses the various phases where diversification can be applied and each step's different techniques. The following subsections provide a detailed summary of these phases and methods.

2.3.1 Implementation Time

N-version programming is the independent generation of $N > 2$ functionally equivalent programs from the exact initial specification [22]. Early work done in N-versioning was aimed toward fault tolerance in mission-critical systems. This approach uses the idea of design diversity where individual teams would implement components separately, design, and have different implementations for similar algorithms. This minimizes the probability of similar errors at decision points; different algorithms, programming languages, environments, and tools are used wherever possible [23].

As explained in [24], the purpose of this approach is those redundant units are intended to compensate for or mask a failed software unit when they are not affected by software faults that cause similar errors at cross-check points. The output of these individual systems is then compared and carried out by selection algorithms or, in most cases, a voting mechanism to derive a

consensus. These mechanisms are used to detect erroneous outputs from the individually created versions. Finally, each version is integrated into the system, becoming a part of the more extensive system.

While N-versioning seems like a practical solution to software diversification and is still reasonably popular within different corporations such as Raytheon, this approach is not without significant drawbacks. First, N-versioning for a given program shows an apparent increase in terms of cost, as a different team is required for every unique version developed. Second, as [7] points out, the logic implemented in one code version may be correct, incorrect, or missing altogether, even though it passes the selection algorithm. Additionally, there is a possibility that faulty but identical results (due to missing logic) may outvote correct results [22]. Finally, N-versioning does not remove ROP gadgets. Instead, another program is created that may or may not have a similar exploit.

2.3.2 Compiling and Linking Time

Software diversification at the compilation and linking stage allows for greater control over how and where we can focus our diversification efforts. As noted in [8], diversifying binaries at this level has three main advantages. The first advantage is information such as symbols and control flow are still intact. This advantage is important because the transformation from source code to object code is a lossy transformation. As a result, of this lossy transformation, perfect recovery of a program's control flow is not generally possible [8]. Second, one compiler mays also support multiple instruction sets and architectures. This also allows transformations to be generalizable and implemented across all compiler-supported architectures. For example, the GCC compiler has a vast variety of hardware models and configurations that are readily available [25]; thus, by adding diversification techniques to the compiler, researchers can take advantage of

various hardware models. Finally, one of the most powerful benefits of compiler-generated diversification is the ability to tap into the compiler's optimizer. This means even after the binary has gone through the optimization phases, diversification still occurs. This would potentially allow us to keep the performance overhead to a minimum.

In [7], the authors present a hybrid approach to diversify software that uses a compiler to embed metadata and a custom toolchain on the client side to achieve diversification. This approach uses a modified LLVM/Clang compiler to embed metadata in the resulting object files. These object files are then updated and consolidated during the linking phase, during which the authors modified the GCC gold linker. In the compilation and linking steps, a new section is added to the *.text* section, which the authors call *.random*. This section is added to each object file and in the final resulting binary. In this work, the authors did not report on results concerning eliminating ROP gadgets. However, the authors did report results on performance and file size. On average, the authors note that they did see a 0.28% increase in performance, which they note is negligible. Regarding file size, on average, authors saw an 11.46% increase in file size. Which, as they note, is a modest size increase.

Additionally, in [26], another approach to diversifying software systems at the compiler level is presented. This approach took advantage of a cloud computing environment to create more variants in parallel rather than sequential. The diversification algorithms employed in this paper were no operation (NOP) code insertion, instructions that do not modify the program processor state, and adjustment of the instruction scheduling. Instruction scheduling is a technique that compilers use to decrease pipeline hazards. In this paper, the authors used a value they called *pnop* to determine if a NOP instruction will follow each instruction.

Regarding selecting how the instruction scheduling would be determined, the authors decided to implement random choice and worst-case instruction scheduling. To evaluate their approach to diversification, the authors collected data regarding security, performance, and file size. The approach taken to measure security was based on a survivor algorithm they developed. Under the survivor algorithm, a gadget is considered "survived" if it appears at the exact memory location after diversification. If the memory location changes, the gadget is considered eliminated. Their results in regards to security show that when using their instruction scheduling algorithm in both worst case and best case, more than 95% of usable gadgets were removed on average. Additionally, when using their NOP insertion algorithm, less than 4 percent of gadgets survived.

When discussing the results relating to file size for all NOP insertion diversification, the authors saw a significant file size increase of 3.9% at the lower bounds and 40% at the upper limit. Finally, regarding performance with the NOP insertion, authors observed an increase in performance ranging from 1.3% to 40%. Some of the degradations are explained with pipeline stalls caused by a bug in their compiler where two NOP instructions were added one after the other. With the second algorithm, slowdowns were observed between 9 and 20 percent, with some binaries having a performance increase well within the margin of error. These slowdowns are not surprising as the compiler places the instructions in a specific way to increase the performance, and rearranging them can lead to suboptimal performance.

While software diversification at the compiling and linking level gives more control over the diversification process, several shortcomings exist. The most obvious one is the availability of a compiler; much of the work presented here has been limited to only open-sourced compilers such as GCC and LLVM. This limitation prevents software diversification from being extended to proprietary compilers such as Microsoft's MSVC. Additionally, this approach requires the

availability of source code, which is not always available for legacy systems. While at the vendor level, it may be hard to imagine that source code would not be available for their legacy systems, there are always exceptions. A specific example of such a case is presented in [27]. Microsoft developers had to hand patch a vulnerability in their equation editor.

While the tools discussed in this section all have shown results that could be useful, unfortunately, they are not open-sourced. Therefore, none of the tools presented will be used in this work, but I found it necessary to summarize the work done at this level.

2.3.3 Installation Time

In these final three sections, I focus on diversification strategies where access to source code is no longer an available. As mentioned earlier, the difficulty in transforming binaries when source code is no longer available relies on the ability to disassemble binaries with and without debugging symbols. While the recursive traversal algorithm is more efficient than the linear sweep algorithm, when disassembling binaries. Factors such as data embedded in the code regions, variable instruction size, indirect branch instructions [28], and encrypted sections contribute to the prevention of perfectly disassembling stripped programs.

However, these limitations have not stopped work in this area. In [29], the authors developed a system of diversifying PE binaries in place. This approach sidesteps the problem of complete disassembly. In this work, authors used IDA Pro to disassemble the binary, ignoring unreachable or unidentified assembly code. Once the disassembly was extracted and converted to an internal representation, different algorithms were used. The first algorithm was an atomic instruction substitution algorithm; in this algorithm, the original instruction was replaced with a functionally equivalent algorithm. For example, given the instruction *cmp bl, al,* the functionally equivalent instruction would be *cmp al,bl*; these instructions both make the same comparison.

However, this instruction can render gadgets unusable. The second algorithm used is an instruction reordering algorithm. In this algorithm, through the implementation of a dependency graph, the authors were able to modify the ordering of the code based on when it was last used and when it was later defined. This had the fortunate side-effect of removing or moving *ret* instructions. While gadgets were not eliminated in most cases, the alternate ordering would shift the gadget around so that attackers would not be able to rely on that gadget. The final algorithm in this work was what the authors called register reassignment. In this algorithm, through the use of a use-def algorithm, values are stored in registers, swapped, and re-assigned. The authors claim they can break ROP gadgets because with registers switched, any gadget that relies on specific gadgets to transfer control flow might jump to incorrect addresses or invalid memory regions. The authors of this paper evaluated their approach using a set of Windows DLLs and reported the percentage of gadgets that were eliminated or deemed broken. They note that their system breaks 80% and removes 10% of valuable gadgets. Their approach also worked in mitigating known exploits such as CVE-2010-2883 [30] when automated ROP exploitation tools such as MONA [10] and Q [11] were used.

Instruction Location Randomization (IRL) was presented in [31]. IRL rewrote binaries so that every instruction is randomized within the process's address space. This approach changes the assumption that programs are loaded and executed sequentially. In this approach, an object dump is used to recover the assembly code for the target program. Once the assembly code has been extracted, the authors use a custom data structure called the *fall-through* map. This data structure contains a set of rules that map assembly instructions to their associated randomized addresses. Once rules are applied to the entire program, jump offsets and addresses are updated. The programs

are run through a process virtual machine. This virtual machine uses the fall-through table to examine and translate instructions before they are executed.

Additionally, code fragments are cached to reduce overhead, and the virtual machine controls the cached code. The authors successfully thwarted tools like ROPGadget [9] to re-create an exploit for CVE-2006-3459 [32]. One such reason this tool could not successfully re-create an exploit is that authors could randomize the location of 99.96% of gadgets through this approach. However, this approach is not without its faults; from a performance standpoint, it incurred performance overheads between 13% and 16%. These overheads can be attributed to the overhead from their process virtual machine compounded with the overhead of their tool. Finally, the last metric recorded was the memory size overhead for each program. As the authors noted, their approach is ineffective, and their rewrite rules can be extensive. The average length of their rewrite rules was 104MB. The increase in length of the rewrite rule is attributed to the authours preferring readability and ease of debugging for this prototype.

As mentioned earlier, the biggest issue is the lack of symbols within the distributed binary. However, this has not stopped researchers from developing tools that can diversify binaries. Marlin [33] circumvents this issue using *Unstrip* [34], a tool designed to help restore symbols to stripped binaries. Once the symbols are recovered, Marlin begins shuffling function blocks based on a random permutation. Additionally, to address the changes in address offsets, Marlin does what the authors call *jump patching*. This process overwrites the original offsets with updated offsets once the shuffling phase is complete. This overwriting, in turn, breaks the assumption required by most ROP-based exploits: the relative offsets and instructions within an application's code are constant. Similar to previous papers in this section, the authors evaluated their approach by presenting the

processing time incurred by Marlin and demonstrating Marlin's capabilities against defending against ROP exploits.

Regarding processing time, the authors did not measure the increase in CPU usage or the memory increase. One reason is that Marlin does not add additional instructions as opposed to other tools I have seen; instead, Marlin rearranges functions in the binary. However, because this tool calculates start addresses and mixes functions every time the binary starts up, an overhead of 3.3 seconds on average was observed in evaluating how Marlin protects diversified binaries against ROP exploits. The authors developed a simple buffer overflow vulnerable application. To create the exploit code, the authors used the popular ROP gadget-finding tool, ROPGadget [9]. When the authors used the exploit code initially developed from the original program, the diversified variant exploits failed. This failure further illustrates the sensitivity of ROP-based exploits and confirms that changes to address layouts are enough to thwart exploits of this nature.

The final approach related to diversification to eliminate ROP gadgets was presented in [35] and is referred to as binary stirring. This approach, similar to the previous method, randomizes the code layout of binaries so that gadgets are found at a specific address in only one instance. This randomization is achieved in two different phases a static phase and a load time phase. In the static step, through the use of a disassembler, the target binary is disassembled. As has been mentioned before, disassembly is not 100% accurate. Therefore, the workaround is to keep a copy of the original binary in a special area in the .text section called *.told*. After creating this special area, a copy of all the bytes that could be disassembled is created. This is done in a section called *.tnew*, again found in the .text section. The original bytes become marked as non-executable to prevent the use of any gadgets that could be found there.

Meanwhile, the disassembled copy is partitioned into basic blocks where jump offsets are calculated and overwritten through a lookup table; this process prevents the program from crashing due to an undefined address. The second phase in this approach uses an external library which loads and executes first. The purpose of this library is to randomly reorder all of the basic blocks in the *.tnew* section. From here, the *.told* section is also updated to point to the new basic block addresses, and this pointing is done because sections of the *.told* section can hold strings or other relevant data. Moving on to the evaluation portion, in terms of gadget elimination, binary stirring was able successfully to render 99.99% of gadgets unusable. It is worth noting that these gadgets were rendered ineffective, not because they were entirely removed from the binary, but because they were no longer located in the same address space. This evaluation was done through the use of three different tools that have been mentioned before: Q [11], Mona [10], and ROPGadget [9] on Linux binaries. The authors evaluated the performance of the SPEC2000 benchmarks after stirring. On average, the SPEC binaries increased by 6.6%, with the Windows program gap exhibiting the worst overhead of 35%.

Although helpful information can be lost by diversifying after compilation, researchers have still found clever ways to partially recover essential sections of a program and diversify what they have to work with. While most of the approaches presented in this section have shown that ROP exploits can be stopped using simple techniques, most do not eliminate gadgets; therefore, these gadgets are still available for the attacker to craft an exploit. This is further confirmed in [12], where authors developed their own method of measuring ROP gadget survival: Bag of Gadgets, where memory location was not considered. Their results show that only a tiny percentage of gadgets are eliminated when memory offsets and addresses are not taken into consideration, as opposed to when they are, for example when methods such as the Survivor

algorithm is used. Additionally, because of these changes, it might be trivial for attackers to modify their exploit code to work for different variants.

2.3.4 Load Time

Load time is when the Operating System's loader begins the process of reading the executable from non-volatile storage (hard drive) and loads it into volatile (RAM) memory to be executed. During this time, shared libraries are loaded onto memory, registers are initialized, and the program begins executing. In terms of software diversification, load time diversification offers the flexibility that diversification can be introduced without the need for source code. However, similar to other approaches mentioned have seen where software is unavailable, we are still limited to disassembling what we can. This section will discuss two different methods in the literature to diversify program load time.

In [36], the authors developed XIFER, a tool that diversifies programs at load time for both the ARM and X86 architecture. XIFER does this by randomizing the memory addresses of the executable and its segments (.text, .init, .data, etc.). In addition to randomizing the memory addresses of the executable segments, the assembly code of the target program is broken down into pieces and randomized within the address space; this is done to prevent memory leak vulnerabilities from disclosing any relevant data or code information. This randomization takes place on the fly before the program executes. While the relocation of a program is similar to Position Independent Executable (PIE) mitigation, the main issue with PIE is that all of the relative offsets within the code remain the same. At the same time, XIFER modifies all of the offsets too. Through the use of a custom library, *libewrite.so*, XFIER begins by intercepting the loading of the executable after libraries have loaded but before the binary starts execution. This customized

library contains its own *.init* section, which loads the necessary libraries and overwrites symbols upon execution.

After the necessary libraries and symbols have been loaded and overwritten, XIFER begins disassembling the program. In this step, the authors use a look-up process to identify any opcodes followed by immediate values or addresses and used as inputs for the re-writing process. If the instruction does not use immediate values or addresses, then these instructions are only seen as black boxes of code and are ignored. This approach, the authors claim, allows their disassembler to be faster than other disassemblers, such as objdump and IDA Pro. Once the program has been disassembled and rewriting instructions have been identified, XIFER begins building the reference graph. This reference graph is similar to a relocation table because it only saves parts of an instruction that point to an absolute or relative address. In this process, all of the identified instructions from the previous step are decoded and saved in a table using a method the authors call *FastDecodw.* This method stores information on how to write back parts of instructions in an assembler-agnostic way; in this step, references to the original instruction is kept. This step is essential as it maintains references to the original instructions even though they might be moved in memory in later steps. After the reference graph has been built, XIFER moves to the transformation phase. In this phase, instruction sequences or individual instructions are broken into chunks, and explicit jump instructions are added at the end of each code sequence, allowing the code to redirect to the new address for the next instruction. This approach, in combination with the reference graph, allows code to be moved to different locations and ensures that jump points connect to the proper blocks of code. The final step in XIFER is the Fixation and Assembly step. In this step, random addresses are given to each piece that has been selected to be relocated. After addresses are assigned to all code sections, the instructions are written back into memory with their

new address; during this step, the references to code and data are updated. This is done using the FastDecode information, and all of the information gathered from the reference graph step.

Finally, after the program has been re-written and the new code sections have been updated, *libewrite.so* is unloaded, and the program begins its normal execution. The authors evaluated their tool on 12 different binaries from the SPEC CPU 2006 suite. The first evaluation was on identifying ROP gadgets; using ROPGadget [9], the authors note that no ROP gadgets were found on the diversified binaries after diversification. In terms of performance, authors measured the runtime overhead and the memory overhead for both architectures supported. The authors claim that runtime overhead was only 5% and 2% for X86 and ARM, respectively. As far as memory overhead goes, authors measured the size of *libewrite.so* and the total increase in binary size. Results show that *libewrite.so* is only 72 kilobytes when loaded, increasing the diversified program's total size by an average of 5%.

A second approach for diversifying binaries during load time was introduced in [37]. In this method which the authors have called Binary Stirring, basic block addresses are determined at load time and can be used on both Windows and Linux binaries with or without symbols. Binary Stirring is broken down into two separate phases: a static rewriting phase and a load-time stirring phase. The target binary is disassembled during the static re-writing phase using IDA Pro. After disassembly, each basic block (contiguous sequence of data with one entry point) is copied into a new section in the binary (*.told*). After all basic blocks have been copied to the *.told* code section, the code goes through a transformation. Using two algorithms, code is transformed into a randomizable representation. As part of this transformation, jump instructions are added to basic blocks so that the code can be partitioned into small chunks that can be randomized during the stirring phase. To maintain the integrity of addresses for jumps, a look-up table is used to track

address mappings. After all the code has been transformed, it is copied to a new section in the binary (*.tnew)*. This new section of code will be executed when the program begins execution. Before the diversified program begins its execution, the load-time stirring phase begins. In this phase, the program is loaded onto memory, and a statically linked library is loaded into memory. This library performs two separate tasks: the first task loads and re-orders all the basic blocks in the *.tnew* section. The second task begins after all the basic blocks have been loaded and re-ordered; the lookup table is used to update all of the mappings stored to ensure that the program jumps to the appropriate code bocks.

Once load-time stirring is complete, the *.tnew* section receives the same permissions as the *.text* section, and execution begins like normal. To evaluate this diversification approach's effects, the authors developed an experiment in which they diversified the SPEC CPU 2000 benchmarks for Windows systems and 99 Coreutils binaries for Linux systems. Their results showed a code size increase of 73% in Windows systems and 3% in Linux systems. Additionally, the authors note that they measured a performance overhead of 4.6% on Windows and 0.3% on Linux applications. The final evaluation the authors measured was the elimination of ROP gadgets. Using ROPGadget the authors report that their approach rendered 99.9% of gadgets unusable (in this context, the authors define unusable if it is no longer in the same virtual address after randomization). The authors note that only pop and ret instructions remain in the exact location. However, the authors do not mention whether any original gadgets exist within variants.

2.3.5 IoT Devices

Unlike other approaches, diversification for Internet of Things (IoT) devices has been relatively limited. Most of the work that can be applied to IoT was done in conjunction with other work presented in previous sections. This limitation can be attributed to authors using a compiler

that supported multiple architectures or expanding their binary re-writing approaches to be robust enough to work on architectures other than x86. Currently, the work presented in this section is only potential research directions that have not been implemented but whose ideas can be applied only to IoT devices. Therefore for thoroughness, this section will present these diversification techniques as they have not been demonstrated in prior sections.

[38] Authors propose two approaches to address security threats in IoT devices potentially. The first approach is to introduce diversification in the OS and APIs used in the IoT device. The main idea is exactly what it sounds like. The entire OS and APIs being used by the IoT device would be diversified and then placed on the device. This approach, in theory, would prevent attackers from injecting malicious code to spy on or manipulate the target system, as the attacker would need to know how to interact with each unique system. As part of preliminary work noted in their paper, authors were able to diversify Linux operating systems and API calls, making it harder for malware to interact with the interfaces. More specifics on their previous work can be found in [38].

The second approach proposed is to apply diversification on communication links among network nodes. This approach aims at making it more difficult for an adversary to gain knowledge of the protocol between the two notes for communication to prevent data packets from being manipulated. Cryptography is a common way to obfuscate the protocol. Different levels of encryption could be employed upon the security need and network capacity [39].

2.4 SUMMARY

This chapter presented an overview of modern binary exploitation techniques, first starting with introducing the buffer overflow before discussing Return Oriented Programming (ROP).

Additionally, this chapter presented several areas where software diversification has been introduced and the approaches used to diversify software. Since software diversification is a research area over twenty years old and has yet to be widely adopted, there may not be a significant understanding of whether diversification will be beneficial in preventing reusable exploits. This work aims to make the analysis of diversification algorithms easier and, in turn, encourage real-world use in mainstream applications. By quantifying these effects and measuring the results, operators can make better decisions in selecting the algorithms used to diversify binaries.

CHAPTER 3: RELATED WORK

3.1 MEASURING THE EFFECTIVENESS OF DIVERSIFICATION APPROACHES

Previous subsections have all discussed and presented approaches and algorithms for diversifying binaries. These approaches have ranged across the software engineering life cycle from implementation, compiling and linking, installation, and load time. The following subsection presents work related to analyzing software diversification's effect on ROP gadget removal and its effect on exploit development, as well as the performance impact that software diversification has.

3.1.1 Diversification on Gadget Removal

In work presented in previous subsections, diversification has been primarily focused on developing diversification engines and diversification algorithms. As noted in [12], researchers do not use a widely accepted methodology to evaluate diversification techniques. This section will discuss different works that have developed systems to evaluate the effectiveness of diversification, both in removing ROP gadgets and mitigating exploits.

In [12], the authors began to explore how diversification techniques affect the available gadgets and their remaining after diversification. The authors developed an approach for evaluating the percentage of gadgets that survived diversification and compared it against an existing method of counting gadget survival. The first approach is the Survivor approach, first presented in [40], which considers gadget sequences and program offsets in its comparison. This approach assumes that a gadget is helpful to an attacker only if the functionality is located at the same address. The second approach, and one that was developed as part of this work, is what the authors have called Bag of Gadgets. The Bag of Gadgets approach is different than the Survivor strategy in that it considers the uniqueness of gadgets, such that even if a gadget is found in two

different binaries at two different memory locations, it would still be regarded as a surviving gadget. The authors used these two methods to measure the amount of ROP gadgets that remain across a set of variants after being diversified. Their results show that by using the Survivor method, diversification can remove anywhere from 90-95% of gadgets in a program. However, this is not the case when the same analysis was done using the Bag of Gadget method. The Bag of Gadget results shows that there is only a slight reduction in gadgets. Additionally, the author notes that this reduction might not be enough to stop code-reuse (ROP) attacks.

In [41], the authors evaluated software diversification's effectiveness in mitigating exploits. In their experiments, authors selected to diversify the DARPA Cyber Grand Challenge (CGC) binaries, as these programs had the Proof of Vulnerability (POV) readily available. This work created one hundred variants per program using the Multicompiler [40] and the Obfuscator-LLVM [42] diversification engines. The authors then ran the POVs against all the diversified binaries and evaluated the number of exploits mitigated using diversification. Results show that diversification was effective against 57.9% of Type 1 exploits, exploits that allow an attacker to gain control of the target program, and was only 12.1% effective against Type 2 exploits, exploits that can cause information leaks.

It should be noted that this does not mean that diversification mitigates all Type 1 vulnerabilities, as most exploited programs require a combination of Type 1 and Type 2 exploits. Finally, the CGC binaries are not an accurate representation of real-world attacks, as these programs are used to demonstrate the presence of a vulnerability. Therefore, they require the minimum degree of work an attacker needs to launch an attack.

In [43], the authors developed a system to measure diversity in terms of code reuse by using near-duplicate detection, an approach that has been used in plagiarism detection programs

[44] and identifying duplicate web pages [45], and symbol table analysis. Using these approaches, authors could define the ground truth regarding code reuse among programs that share code. This includes executables that share functions from statically-linked libraries. In their experiments, the authors diversified a wide array of binaries. More specifically, they diversified: GNU core utilities, Docker Images, Ubuntu packages (32 and 64-bit packages), and Microsoft's Malware Challenge. Results indicate that strategies implemented by diversification compilers are only marginally successful, and while they do introduce considerable differences from non-diversification approaches, similarity remains significant. While in this work the authors did not analyze surviving gadgets on diversified variants, their work does propose that future work would correlate near-duplicate detection with exploit prevention.

Finally, in [46], the authors developed a case study and analyzed the gadgets found before and after the diversification introduction. The authors began by measuring the number of gadgets found in non-diversified variants and classifying these gadgets based on their behavior in the set of GNU core utilities, a group of commonly used Linux utilities. Following this measurement and classification, they diversified GNU core utilities and analyzed the difference in the number of gadgets and the change in where gadgets fall into each category. In this work, the authors observed an increase in the total number of gadgets in diversified variants. Furthermore, the authors also note that because of this increase, there was an increase in all gadget categories. Similar to other work, however, the authors did not analyze if this increase in gadgets allows an attacker to develop an exploit that can be re-used. Nor did they explore if new exploits could be developed with the added gadgets that would not be possible without diversification.

3.1.1 Diversification on Binary Performance

In [1], my co-authors and I designed and implemented an analysis system that facilitates the diversification of binaries using the Amoeba diversification engine [3]. This system presented in that work was designed, implemented, and released to analyze the performance of diversified binaries and how they are fair compared to the original. The tool developed was released as an open-source analysis system that collects and visualizes the metrics associated with binaries diversified using Amoeba [3]. However, as this dissertation will discuss in chapter 5, the system initially presented was expanded to include new components. Additionally, as part of this original work, a case study was conducted to illustrate the performance impact associated with diversification and the total number of shared ROP Gadgets.

In the original experiment, all binaries were diversified 20, 30, 40, and 50 times each using the nine diversification algorithms provided by Amoeba. The final algorithm, basic block flattening, was not used due to non-responsiveness or failures when used with over 30 iterations. After completing the diversification process, Perf executed and recorded performance metrics for all binaries.

As part of the original study, the first binary diversified was Shred, a Linux utility program that overwrites a file to hide its content. Shred, at the time, was selected primarily for its functionality of writing and re-writing to disk. Shred was executed on a one-gigabyte file with default arguments for this study. Figure 24, found in Appendix C, displays the original bar graph of the results observed. For these figures, the x-axis is the name of the binary. The first number represents the number of diversified iterations, and the second represents the diversification algorithm used. The y-axis represents the number of CPU cycles taken. For readability, a black line has been added to represent the results for the original binary.

The original version of Shred completed execution at around 13,000 CPU clock cycles, with most diversified variants staying within the same range peaking at approximately 15,000 CPU cycles. However, these results show two diversified binaries that stand out due to their significant peaks. These binaries, Shred.20.005 and Shred.30.005 required roughly 29,000 cycles and 18,000 cycles, respectively. Both of these binaries were diversified using a function reordering algorithm. The only main change between the two was in the number of diversification iterations, Shred.20.005 went through 20, and Shred.30.005 went through 30. A special note is that when Shred was diversified 40 times with the same algorithm, it performed slightly better than the original binary. This performance improvement could mean that for this binary, more diversification iterations could improve its performance.

Regarding comparing the total number of ROP gadgets between the diversified binaries and the original, Figure 25, found in Appendix D, presents the results from that experiment. Again, the x-axis is the name of the binary. The first number represents the number of diversified iterations, and the second represents the diversification algorithm used. The y-axis represents the total number of gadgets.

As observed, the original Shred contained 1,081 unique gadgets. In addition, the figure shows that most of the diversified binaries had some ROP gadgets eliminated, with most averaging around 990 distinct ROP gadgets. However, as observed in that work, three binaries removed more ROP gadgets than the rest. These binaries are Shred.20.004, Shred.30.004, and Shred.40.004. These variants contained 756, 757, and 760 unique ROP gadgets. All three of these binaries were diversified using an instruction replacement algorithm. This elimination of ROP gadgets could be because diversification may have broken down complex instructions during the replacement phase. Overall, however, there was not a significant reduction in ROP gadgets.

The second binary discussed in the original work was CP, a Linux utility program that copies files and directories. CP was selected for its functionality of writing bytes to a disk. Similar to the Shred, CP was executed on a one-gigabyte file. Figure 26, found in Appendix E, details the results when analyzing the performance impact.

In that experiment, the original CP completes the copying operation in roughly 2,500 CPU cycles, with most diversified variants staying within the same range peaking at around 15,000 CPU cycles. Also similar to Shred results, two binaries stand out due to the significant peaks. As was the case with Shred, these binaries were CP.20.005 and CP.30.005, which took longer than 7,000 cycles and 4,000 cycles, respectively, to complete. Like Shred, these binaries were diversified using the function reordering algorithm. However, unlike in the previous binary, when diversified for 40 iterations, CP did not do better. In fact, the results presented show it took around the same time as the original non-diversified binary to complete. The results show that function reordering improves performance with CP, unlike with Shred. For instance, when diversified for 20 iterations using the basic block split algorithm, CP appears to have a slight performance improvement. When comparing the total number of ROP gadgets between the diversified binaries and the original, the original CP contained about 2,000 unique ROP gadgets. Figure 27 shows the plot details comparing the total number of ROP Gadgets.

As with Shred, the total number of ROP gadgets between the original and the diversified binaries dropped. CP variants that were diversified 20, 30, and 40 times using the instruction replacement diversification algorithm showed the best results regarding the total number of unique gadgets, with a reduction of about 500 ROP gadgets compared to the original. These results can be seen in Figure 27, found in Appendix F.

The final binary discussed as part of the case study in the original work was ls. Ls is a Linux utility program that lists directory contents. ls was selected to show how diversification can affect binaries that traverse directories. To gather performance metrics, ls displayed all files on the system in that experiment with the –R flag. Figure 28 details the results recorded.

The original ls completed executing in about 3,500 clock cycles. However, unlike in other experiments, the peaks that stand out the most in this experiment originated from the same diversification algorithms. These algorithms are control flow branch diversification at the function level and basic block splitting, with the peak being: control flow branch diversification with ten iterations. This result could show that depending on the type of binary and the diversification algorithm used could also significantly impact the binary's performance. Figure 28, found in Appendix G, displays the results when comparing the total number of gadgets found before and after diversification.

Once again, as with the other binaries observed that work, there is a reduction in ROP gadgets using software diversification. Furthermore, similar to previous examples, the most significant reduction comes from the instruction replacement algorithm. These results can be seen in, Figure 29 found in Appendix H.

The results presented in that work show that different algorithms potentially impact binaries differently. Those results also show it is crucial to understand these impacts, especially when working with limited and constrained systems such as the Internet of Things (IoT) and the Internet of Battlefield of Things (IoBT) devices. Chapter 5 will discuss the expansion of this system as part of this work.

3.2 SUMMARY

This chapter introduced previous work in quantifying software diversification's impact. Apart from work done in [2], [3], and [4], there has still been little work done regarding quantifying the effects and limitations associated with software diversification. This chapter also presented my previous research on understanding the impact of software diversification algorithms from a performance aspect. The next chapter will discuss the research questions associated with this work.

CHAPTER 4: RESEARCH QUESTIONS

4.1 RESEARCH GOAL

Software diversification has been presented as viable mitigation to Return Oriented Programming (ROP) exploits. Unfortunately, most of the work presented in the literature is primarily focused on diversification algorithms and where these diversification algorithms can be introduced in the software development lifecycle. As a result, there has been a lack of research into the effectiveness of diversification techniques [5] [2] in quantifying the effectiveness of diversification and how these techniques affect ROP chain re-use and ROP chain development.

This work aims to analyze the impact of software diversification on the development of ROP-based exploits. This work also seeks to define methods to quantify diversification's effect on diversified binaries. This work also presents a selector tool developed to assist operators in analyzing, visualizing, and selecting the appropriate diversification algorithm, given their preferences. Through this research and the development of this methodology, the research community can begin to understand the impact diversification has on ROP exploit creation. With the development of quantifiable methods, researchers can analyze the benefits, trade-offs, and side effects resulting from software diversification. This work defines side effects as an increase in total execution time, CPU computation time, and additions of new ROP gadgets.

While previous work has looked at the percentage of shared gadgets between non-diversified and diversified binaries, these works did not analyze whether the shared ROP gadgets between variants are enough for an attacker to develop a shared exploit. One of the outcomes of this work is to create a case study to analyze and understand that impact. Through the use of this case study, this work aims to show how effective software diversification algorithms are in protecting against commonly used ROP chains. Additionally, this work seeks to know if it is

feasible for an attacker to create a new exploit with the ROP gadgets that survive diversification or with the gadgets added during the diversification process. Based on the results in these areas, this work identifies and develops several metrics to measure the impact of diversification algorithms.

The research questions associated with this research are as follows:

RQ1: How effective are software diversification algorithms in preventing attackers from using previously crafted or developing new ROP-based exploits?

RQ2: What are the primary criteria to consider in determining the efficacy of diversification algorithms in preventing exploit re-use and development?

RQ3: What is the appropriate set of metrics to quantify the efficacy of software diversification?

In addition to answering these research questions, this work also designs and creates a selector system that will assist operators in identifying what software diversification algorithms would be best for their use case.

This work begins by analyzing software diversification's impact on ROP-based exploit development to achieve the research goal. This work is then followed by then identifying the appropriate set of criteria necessary to quantify software diversification and developing quantifiable metrics based on these criteria to measure software diversification's impact on binaries. Finally, by creating a selector system, this work will allow the end user to identify diversification algorithms that best meet their needs while considering the metrics developed and any performance impact associated with diversification.

CHAPTER 5: METHODOLOGY

5.1 EXAMINING THE IMPACT OF SOFTWARE DIVERSIFICATION ON EXPLOIT DEVELOPMENT

As part of this work, a case study was developed to better understand software diversification's impact when generating exploits and its role in allowing an attacker to re-use an exploit. This section describes the methodology created to understand software diversification's impact on exploit development and answer R1: How effective are software diversification algorithms in preventing attackers from using previously crafted or developing new ROP-based exploits?

5.1.1 Experimental Overview

I developed a case study that uses real-world binaries with known and documented exploits to analyze the impact of different software diversification algorithms on ROP chain generation. For this work, it was important that the case study developed use real-world binaries outside the traditional GNU Coreutils dataset. While the GNU Coreutils dataset is appropriate to measure diversification's impact, that dataset does not suffer from many known and exploitable vulnerabilities.

The following topics discuss in depth the approach for Binary Selection, Diversification Engine Selection, and the ROP gadget finder toolset.

5.1.1.1 Diversification Engine Selection

I decided to use Obfuscator-LLVM [47] as the main diversification engine in this work. While the official version of Obfuscator-LLVM is built upon LLVM version 4.10, I found and

built a copy using LLVM version 10.0. The decision to use an open-source compiler was beneficial and due to several reasons. First, the decision to use an open-source diversification engine for simplicity and replicability. While it would have been better to use multiple diversification engines, most software diversification engines that we analyzed have been closed-source and therefore cannot be used for our analysis. Second, the ability to diversify at compile time allows for greater control of where the diversification happens. In previous work that I have done in the software diversification area [6], I utilized binary-level diversification engines. In that work, runtime issues were raised in which diversified variants have had unwanted segmentation fault errors due to the consistent assemble and disassemble process from the diversification process. These errors are due in part to the complexity of this disassembly re-assembly phase, Disassembly is still a complex problem to solve and an active area of research. As a part of this work, I generated and analyzed exploits for diversified and non-diversified variants, documented the changes in exploits, and investigated if the shared gadgets found after diversification allow an attacker to create a re-useable exploit.

Another motivating decision was widely accepted in previous works and the literature, as there has been a wide variety of work that uses LLVM and studies that have used O-LLVM in the diversification literature.

Finally, by using Obfuscator-LLVM, I can take advantage of several out-of-the-box transformations. These transformations are also widely accepted in diversification and obfuscation literature. The transformations supported and a brief description of these transformations are as follows:

Instruction Substitution: This obfuscation technique replaces binary operators such as addition, subtraction, or Boolean operations with a functionally equivalent set of procedures to maintain the same functionality. [43]

Control Flow Flattening: This obfuscation technique works by flattening the control flow graph of the binary. This is done by modifying basic blocks and putting these blocks that were originally at different nesting levels next to each other. [43]

Bogus Control Flow: This obfuscation technique modified the function control flow graph by adding a basic block before the current basic block. The new basic block contains an opaque predicate, which will make a conditional jump to the original basic block. [43]

While these transformations may seem limited, Obfuscator-LLVM allows combining different algorithms, allowing for a total of seven various mutations.

5.1.1.2 Binary Selection

Because I am interested in seeing software diversification algorithms' impact from an exploit development standpoint, I decided to move away from using the GNU Coreutil binaries as our dataset. This decision is a change from previous research, which looked at gadget survival primarily using GNU Coreutils as the primary dataset. While in this work, I diversified GNU Coreutils and analyzed the effects of software diversification algorithms on Coreutils as a baseline for experimentation since I could not develop complete working exploit chains using Coreutils. Additionally, Coreutils might not accurately represent the general population of binaries available.

Furthermore, as known from prior research aside from touch and date, not many vulnerabilities have been discovered in GNU Coreutil binaries. Therefore, using ExploitDB, I searched for binaries that best met my selection criteria. ExploitDB is an archive of proof-of-

vulnerabilities (POV) and proof-of-concept (POC) exploits which publicly documents previously identified vulnerabilities in software systems. By using ExploitDB with its search functionality, I could quickly identify a dataset.

The selection criteria were as follows: First, the program should be vulnerable to a buffer overflow. Meeting this criterion is critical because, as mentioned in Chapter 2, ROP exploits are the successor of buffer overflow vulnerabilities and therefore need a buffer overflow vulnerability to overwrite the return address and hijack program execution. The second criterion was the program should have a sample exploit associated with it; this criterion was primarily used to confirm a vulnerability in the program. The final criterion was the vulnerable binary should have source code available, either through ExploitDB itself or through the vendor. This criterion was just as important as our first one because, as I are using a compiler-based diversification engine, a source code was needed to compile and apply the transformations. Using these criteria, nine binaries were identified and used in this work.

The identified programs and a brief description are as follows:

- **PDFResurrec**t- A tool aimed at analyzing PDF documents.
- **DNSTracer**- A tool to determine where a given DNS receives information.
- **MP3Info**- A tool used to read and modify the ID3 tags of MP3 files.
- **SIPP**- A Sip protocol test tool.
- **Netperf**- A benchmark tool that can be used to measure the performance of different types of networking.
- **LamaHub**- A multi-platform NMDC Protocol server.
- **yTree**- A tool for working with merger tree data from multiple sources.
- **Mcrypt**- A replacement for the Linux crypt command.

- **Crossfire-server**- An open-source, cooperative multiplayer RPG and adventure game.

Apart from these nine binaries, OpenSSL, an open-source software library used in applications that use secure communication protocols, was identified and also used. The decision to use OpenSSL was due to the widespread Heartbleed vulnerability, which also used a ROP chain to leak private information.

5.1.1.3 ROP Gadget Analysis Toolset

The best and most effective ROP gadget finders are built using the Galileo algorithm [48]. The Galileo algorithm introduced in [48] has been shown to be an effective method for identifying gadgets that can be used to generate a ROP chain; therefore I selected to use the Galileo algorithm as our analytical toolset.

The Galileo algorithm begins by building a tree of possible ROP gadgets. This algorithm first identifies a 'c3', the operation code for the 'ret' instruction, hex values in the binary. After identifying a 'c3', the algorithm then tries to construct a ROP chain by working backward to determine if the instructions before the 'c3' operation code can be used to create a ROP gadget. This algorithm's main advantage is that it allows tools to identify more ROP gadgets than traditional gadget-finding tools that only look at gadgets at the end of a function.

Two separate tools were identified that use this algorithm. Both tools internally use the Galileo algorithm and have been used extensively in the literature and in the hacking community. The first tool identified was ROPPER [49], while the second was ROPGadget [9]. For this work, however, ROPPER was the best tool, for several reasons. First, ROPPER supports searching for *syscall* gadgets even if the gadget is found in unaligned memory. ROPPER first looks for a useable *syscall* gadget in the binary; if one is not found, it begins looking for the opcode '0f05', *syscall*

opcode, in the entire binary. These opcodes could be seen as part of a more extensive instruction that might be harder to diversify away, as shown below in Figure 6, and thus be identified as ROPPER as a valid *syscall* gadget. For readability, I have highlighted the opcode associated with the *syscall* gadget.

```
8b 0f:             MOV param_4, dword ptr[param_1]
05 fc 0d c7 bb:    ADD eax, 0xbbc70dfc
```

Figure 6:Unaligned *syscall* Gadget Found by ROPPER

The second reason ROPPER was selected was its scriptable Python API, which allows a more straightforward method to analyze gadgets. Finally, ROPPER was chosen because it allows for the creation of two different exploit chains, *execve*, a system call that executes a program as specified by the pathname, and *mprotect*, a system call that allows for the changing of protection on the specified page of memory. While ROPGadget does have some support for ROP chain generation, in our pre-analysis ROPPER was discovered to be the superior tool as it generated shorter and more complete *execve* chains than ROPGadget.

5.1.1.4 Experimental Design

To analyze software diversification's impact on the generation of ROP-based exploits, I developed an experiment using the programs and tools described earlier in this section. As part of this experiment, all programs were compiled in two different ways. First, programs were compiled without any transformation applied, which I call original; this compilation was done to have a non-modified version of the program, which would serve as our control dataset. The second way binaries were compiled was with each of the diversification algorithms ten total times and

compiled with all possible combinations. These combinations resulted in a total of seven different transformations, which has been short-handed:

- Instruction Substitution (sub)
- Control Flow Flattening (fla)
- Bogus Control Flow (bcf)
- Instruction Substitution and Control Flow Flattening (sub fla)
- Instruction Substitution and Bogus Control Flow (sub bcf)
- Bogus Control Flow and Control Flow Flattening (bcf fla)
- All three transformations (sub bcf fla)

This approach allowed for a total dataset of 700 compiled programs, with 70 unique variants for each program.

After compiling all the programs, I collected the total number of gadgets and the total number of functions for all of the binaries in our dataset. I also created two different ROP exploits for all variants and the original binaries. The following section will discuss how the data collected was used to generate the results for this work.

5.2 QUANTIFYING SOFTWARE DIVERSIFICATION

This section identifies the criteria to answer R2: Identify the primary criteria to consider in determining the efficacy of diversification algorithms in preventing exploit re-use and development. Additionally, this subchapter proposes methods to answer R3: What is the appropriate set of metrics to quantify the efficacy of software diversification?

5.2.1 Identifying the Appropriate Quantification Metrics

As mentioned previously, in the research, there have not been methods or models proposed to quantify the impact of software diversification. As a result, there is currently no way to understand the strengths and trade-offs associated with diversified binaries, primarily on how diversification affects an attacker generating new exploits. However, in [2], the authors offer insight into possible criteria. Most notably, they point out that an ideal diversification scheme should satisfy three objectives: Attack Resistance, Reducing Exploit Re-use, and Resistance to Reverse Engineering. This work defines the Reducing Exploit Re-use as the change in the exploit code. Therefore, I have renamed this objective to Exploit Complexity. This name change originates from the work conducted earlier and personal experience developing exploits. While analyzing gadgets used to generate an exploit, I observed that in most cases, ROP gadgets become longer and more complex. Thus this work proposes the following three criteria to consider in determining the efficacy of diversification algorithms:

- Attack Resistance
- Exploit Complexity
- Resistance to Reverse Engineering

In the following sections, this chapter will discuss various metrics used to quantify the effectiveness of software diversification algorithms.

5.2.2 Quantifying Attack Resistance

As described in the previous section, based on the literature review conducted, this work identified three primary criteria for determining the efficacy of diversification. This section discusses and proposes a metric to quantify Attack Resistance.

This work proposes calculating the difference in ROP gadgets between the diversified variants and the original binary to measure Attack Resistance. Equation 1 details the formula used to calculate the Attack Resistance score.

$$Attack\ Resistance = \left(1 - \frac{Gadgets\ in\ Variant}{Gadgets\ in\ Original}\right)$$

Equation 1: Attack Resistance Formula

Calculating Attack Resistance using Equation 1 originates from my previous work and observations. In the work presented originally in [1], my co-authors and I observed that in some cases, Amoeba [6], the diversification engine used in that work, reduced the total number of gadgets, while in other instances, I observed an increase in the number of ROP gadgets. Additionally, when analyzing the published literature, most authors claim that their proposed algorithm is a superior approach to diversification because it introduces randomization without increasing the total number of ROP gadgets in the diversified binary or it removes ROP gadgets altogether. This claim is significant because, as noted in [2] and [7], the overhead associated with developing a successful or reusable exploit is low, as most ROP-bussed exploits only require a small subset of gadgets. Thus an increase in the number of gadgets could lead to an attacker developing a reusable exploit.

Using the Attack Resistant score described, an analyst can use diversification to introduce randomness and security into the ecosystem while ensuring that ROP gadgets introduced are kept to a minimum. Maintaining an Attack Resistance score close to one or lower than one provides security without risking the possibility that ROP gadgets can be used or re-used to launch a successful exploit.

5.2.3 Quantifying Exploit Complexity

The second criterion identified as part of this work is Exploit Complexity. As the proposed method for quantifying the Exploit Complexity score, this work counts the total number of clobbered registers, any register over one modified in a specific gadget, as a method to calculate Exploit Complexity. Figure 7 shows an example of a ROP gadget that "clobbers" more than one register.

```
pop rax; pop rbx; pop rbp; ret;
```

Figure 7: Example of a ROP gadget that clobbers registers

In the figure above, the primary ROP would be **RAX,** as it is the gadget of interest to the attacker. However, as can be seen, two other registers are modified within the ROP gadget (**RBX** and **RBP**); the unwanted modifications of these gadgets could have undesirable consequences for the attacker.

As discussed in previous sections, ROP gadget-finding tools such as ROPPER and ROPGadget are excellent for finding gadgets because of the use of the Galileo algorithm, but they fall short when generating exploits. More sophisticated exploit development tooling, such as ANGR ROP [8], PEASE [9], and Majorca [10], utilize a combination of symbolic execution or constraint-solving frameworks such as Z3 [11], which allow them to generate far superior ROP chains. Because these tools use these frameworks, they consider clobbered gadgets in determining the feasibility of creating an exploit. As it could be the case due to the consistent clobbering of critical registers, a ROP exploit would be unfeasible.

Furthermore, tools like Majorca also utilize a scoring system to determine the "fitness score" of ROP gadgets. As part of this scoring system, clobbered registers are a critical component of the fitness score; the more clobbered registers there are, the lower the fitness score is for that specific ROP gadget. Additionally, the more registers clobbered within a ROP gadget can mean a potential increase in the length of the gadget. As discussed in [12], an increase in ROP gadget length leads to a degradation in the quality of ROP gadgets and unwanted side effects.

For a diversification algorithm to be effective using the metric described, the expectation would be a high Exploit Complexity score. An Exploitation Complexity score higher than one found in the original binary would signify that the quality of the gadgets would degrade due to more registers getting clobbered, making it difficult for an attacker to create an exploit. A high Exploitation Complexity score would also mean that ROP gadgets will have unwanted side effects. Likewise, a diversification algorithm with a low Exploit Complexity score would signify that there are more singleton gadgets, that is, more gadgets that modify one and only one register, which could lead to ample space for exploitation.

5.2.4 Quantifying Resistance to Reverse Engineering

The final criterion identified as part of this work is the Resistance to Reverse Engineering. This work proposes the summation of the McCabe Cyclomatic Complexity [56] for each function in the diversified binary as the method to calculate the Resistance to Reverse Engineering score.

The McCabe Cyclomatic Complexity is well known for its use in software engineering to measure software complexity. However, this formula is also used extensively in the software obfuscation world to calculate the quality of obfuscation algorithms. In [12], the authors present measures to evaluate the strength of obfuscation techniques. One such measurement is Cyclomatic

Complexity, which, as the authors present, falls into the control-flow-based metrics. Furthermore, in more recent work in a literature review on obfuscation published in 2021, [13] Cyclomatic Complexity is still a popular measurement as it is among the three highest frequency topics.

Additionally, Cyclomatic Complexity has not only been used in the obfuscation literature but also as a method to measure the increase in complexity of modern malware. The authors in [14] and [15] explain how the Cyclomatic Complexity score is still widely accepted for calculating software complexity. They detail how they use McCabe's Cyclomatic Complexity to measure changes in modern malware's sophistication over the past thirty years.

Similar to the Exploit Complexity score, for a diversification algorithm to be effective using the metric described, the expectation would be a high Resistance to Reverse Engineering score. A higher Resistance to Reverse Engineering score would result in it possibly taking a reverse engineer or an attacker longer to identify the vulnerability in the diversified binary. A high Resistance to Reverse Engineering score would also mean that even modern vulnerability-finding tools, such as Fuzzers, would have difficulty finding a crash, which could mean a vulnerability.

5.3 IMPLEMENTATION OF THE SELECTOR SYSTEM FOR DIVERSIFIED BINARIES

This section discusses the implementation details of the selection system developed for selecting and visualizing the appropriate diversification algorithm. This selection system expands the work presented in [6] by my co-authors and myself.

This subchapter is organized as follows. First, the technologies used to implement the selector system are described, followed by the components that make up the selector system. Finally, a case study illustrates the use of the selector system to identify the appropriate diversification algorithm and the performance impact that algorithm has.

5.3.1 Choice of Implementation Platform

To facilitate the development of this selection system, several tools were utilized. These tools included a programming language that supports publicly available and commercial third-party libraries for binary analysis, data plotting, and multiplatform execution. Additionally, these tools have been developed with the flexibility to run in an integrated development environment (IDE) or inside a code editor and operating system where this system can be run.

To implement each component in the system, this work used Python version 3.10. This decision is primarily due to Python's robust libraries, as well as the growing popularity of Python in modern binary analysis tools such as Ghidra [16], Binary Ninja [17], and IDA Pro [18]. Section 6.2.3 will discuss in more depth why these tools were necessary.

This selection system was initially developed in Visual Studio code primarily because it is free. In addition, Visual Studio Code supports a robust extensions marketplace, allowing for the installation of lightweight Python linting, ensuring the code written adheres to best code practices. Finally, while Visual Studio Code was the primary development environment, several components were developed in PyCharm, a Python IDE.

This system runs on the Ubuntu version 22.04 Long Term Support (LTS) distribution of the Linux operating system. This new operating system is an upgrade from the prior system that utilized Ubuntu's outdated 12.04 LTS version. The decision to use this updated version of Ubuntu is to maintain the tools as updated as possible to allow for continuous development. Additionally, the diversification engine and other libraries require an updated platform. Additionally, because Ubuntu is open source, it can be installed on any computer without a license. Also, most of the binary analysis tools used in this work interface easier on Linux systems than on other systems.

5.3.2 System Components

As mentioned earlier in this chapter, the selection system was built as an extension of work previously published by myself and my research group. This work was designed and built to be automated, configurable, and allow for repeatability. In the updated state, this system provides analysis of diversified binaries using any of the diversification algorithms and combinations currently supported by Obfuscator-LLVM.

This system's second goal is to include a robust data-capturing mechanism to quantify the impact based on the criteria and quantification methods presented in chapter 5 and record binary performance. The data collected is used along with user-specified input as to what criteria they are most interested in analyzing or maximizing.

Finally, the system must provide the ability to visually represent the data collected to facilitate the comparison of algorithms and visualize the performance impact associated with each algorithm. The following sections will describe each of the components in further detail, starting with the high-level design of the system.

5.3.3 High-Level System Design

This system currently consists of several components: the main execution engine, the diversification component, an analysis component, which has since been updated from previous work [1] to include the calculation of the metrics described in section 5.2, a visualization component, and a newly built algorithm selection component, as shown in Figure 8.

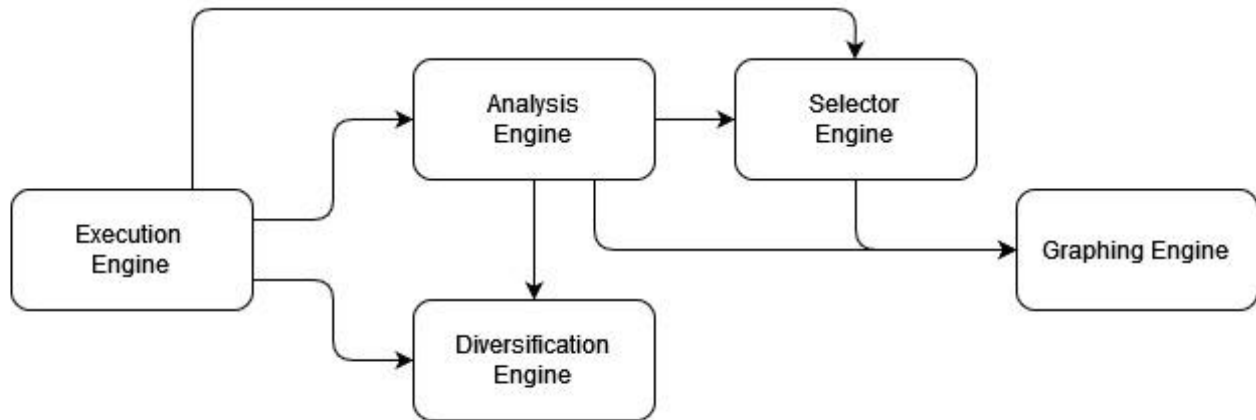

Figure 8: Selection System Architecture

The primary responsibility of the execution engine is to start this system's diversification and analysis engines. Once the diversification engine creates diversified variants, it stores the diversified binaries in a specified directory. The analysis engine then collects metrics from the diversified binaries, saving these results to a comma-separated value (CSV) file. Finally, the graphing engine will use the results to produce interactive charts.

The system design utilizes a plug-in-based architecture that allows components to be swapped out and thus enables the user to extend and modify the standard functionality, as was done as part of this work.

5.3.4 Execution Component

This system uses an execution engine to automate the diversification, analysis, selection, and graphing process. The execution engine starts by reading a configuration file. This configuration file contains the number of variants to create, the name and path to the program's source code, the diversification algorithms to be used, the weight for each of the quantification criteria that users are interested in maximizing criteria 1 (C1) which represents Attack Resistance, criteria 2 (C2) which represents Exploit Complexity, and C3 criteria 3 (C3) which represents

Resistance to Reverse Engineering. The final field in the configuration file is the performance metrics to record. In this case, these metrics were related to CPU clock time.

After reading and parsing this configuration file, the execution engine provides the path to the program source code, the number of variants to create, and the names of the diversification algorithms to the diversification engine. Afterward, all diversified binaries are stored in a separate folder for later use. The execution engine forwards the set of performance statistics to be recorded and the path to the diversified binaries to the analysis component.

5.3.5 Diversification Component

In the first iteration of this system, Amoeba [6] was used as the diversification engine. This engine has since been replaced with Obfuscator-LLVM. This replacement was a widely needed update, as Amoeba, a binary-level diversification engine, only supported x86 binaries with debugging symbols, whereas Obfuscator-LLVM is a compiler-based diversification engine. This addition allows for the compilation and diversification of programs to any architecture supported by Obfuscator-LLVM, which will allow for further analysis of how diversification affects other architectures.

The execution engine parses the following from the configuration file: the number of variants to be created, and the diversification algorithm(s) to apply. Those values along with the path to the target binary are sent to the diversification component. The diversification component initiates Obfuscator-LLVM, ensuring that the program is compiled with the correct diversification flags. During the compilation process, variants are saved in their respective folders. After the compilation phase, the file path where all diversified binaries are stored is sent to the analysis component.

5.3.6 Analysis Component

A significant goal of this selector system is to quantify the impact of diversification algorithms based on the criteria and methods presented earlier and analyze the performance of diversified binaries. This work utilizes ROPPER [19] to quantify two of the three scores. The decision to use ROPPER to find ROP gadgets is due to its gadget-finding algorithm. As previously discussed, ROPPER utilizes the Galileo algorithm to find gadgets, allowing ROPPER to locate more gadgets than its counterparts. Finally, because ROPPER also includes a robust and powerful Python API, it provides for scriptable analysis of the gadgets found.

First, ROPPER collects and counts the total number of gadgets for all variants and the original binary. The information gathered from ROPPER is used to calculate the attack resistance score, using the approach described in section 5.2.1 for both the original binary and diversified variants. Then, using ROPPER's API, the analysis component calculates the exploit complexity score. After the analysis component calculates these two scores, they are saved to a CSV file for later use by the algorithmic selection component.

This system uses Binary Ninja to calculate the resistance to reverse engineering score as part of the analysis component. However, other disassemblers, such as Ghidra or IDA Pro, can be used because of the plug-in architecture. The decision to use Binary Ninja was primarily due to known issues with Ghidra's binary analysis phase. During the analysis process, Ghidra sometimes never finished analyzing a binary, as was discovered in an earlier iteration when Ghidra was initially used to calculate the resistance to reverse engineering score.

After Binary Ninja calculates the resistance to reverse engineering scores for both original binaries and diversified variants, the system writes these scores to the same CSV file that contains the attack resistance and exploit complexity scores.

Finally, to achieve the second goal of this system and analyze the performance of diversified binaries, Perf, the Linux performance tool, is used to collect metrics. Perf can measure and record information such as CPU usage, process memory usage, and power consumption. Perf executes the diversified and non-diversified binaries using the program arguments provided by the execution engine. After Perf generates the results, the system stores the generated results in a directory. The graphing engine then reads these files to generate the plots of the results.

5.3.7 Algorithm Selection Component

With the development of the algorithm selector component, the system's final goal is to recommend the best diversification algorithm to the end user is achieved. This component parses the CSV file generated by the analysis component using pandas, a data science library written in Python. After parsing the CSV file, the algorithm selector normalizes Attack Resistance, Exploit Complexity, and Reverse Engineering Resistance scores for every binary in the CVS file. This normalization uses the normalization formula shown in Equation 2, where X is the non-normalized value for each quantified impact value.

$$X_{norm} = \frac{(X - X_{min})}{(X_{max} - X_{min})}$$

Equation 2: Normalization Formula

Following the normalization process, the selection component calculates a selection score for every binary using the criteria C1, C2, and C3 from the configuration file. These criteria represent the percentage of importance the user wants to give to each quantified value generated from the analysis component. Equation 3 details the formula used to calculate the selection score.

$$selection\ score = (\ C1(Attack\ Resistance_{norm}\ ) + C2(Exploit\ Complexity_{norm}) + C3(Reverse\ Engineering\ Resistance_{norm})\ )$$

Equation 3: Impact Score Formula

After all binaries in this dataset have a selection score, the algorithm selector component sorts the resulting scores from highest to lowest. The analysis component selects the first two algorithms with the highest selection score. The entire dataset is then sent to the visualization component to graph how each algorithm compares.

5.3.8 Visualization Component

To assist in visualizing the results generated from Perf and the analysis component, Plotly [20] is used. Plotly is an open-source library that is a simple yet powerful graphing module for Python that supports various types of interactive plots. In addition, Plotly produces its graphs in an HTML output, allowing an easy view of the graphs locally or on a web page hosted on a server.

The visualization component utilizes the CSV files generated by Perf during the analysis phase, the output generated by the algorithm selector component, and plots the data for each binary in the CSV file. In this work, the graphs displayed to the user are as follows. Bar graphs are used to show the results from the performance metrics generated from the analysis components. In contrast, box charts display the best diversification algorithm for a user. Using box charts to show the best diversification algorithm, users can see how algorithms compare and the range of impact of each diversification algorithm.

5.4 SUMMARY

This chapter discussed the overview of the case study to analyze the impact that software diversification has in terms of preventing attackers from developing exploits or re-using previously crafted exploits. Additionally, this chapter discussed the identification of criteria needed to quantify the impact of software diversification and proposed methods to quantify the effect. Finally, this chapter discussed the implementation details of the selection system for deciding on the best diversification algorithm based on user preference.

CHAPTER 6: RESULTS AND OBSERVATIONS

6.1 IMPACT OF SOFTWARE DIVERSIFICATION ON EXPLOIT DEVELOPMENT

The main goal of software diversification is to prevent an attacker from developing a working master exploit that can be used across all systems. Most of the work done in understanding how diversification affects ROP exploits has only examined the change in gadgets and has not studied if the surviving gadgets or gadgets added as part of the diversification process are enough to allow an attacker to create a successful exploit. This section discusses the work done on examining compiler-based software diversification's impact in generating and creating ROP exploits that can be used.

6.1.2 Gadget Count

As previously mentioned, the first analysis analyzed the relationship between the number of functions in the binary and the total number of gadgets. Figure 9 shows the relationship between the number function in a binary for the original (non-diversified) variants generated.

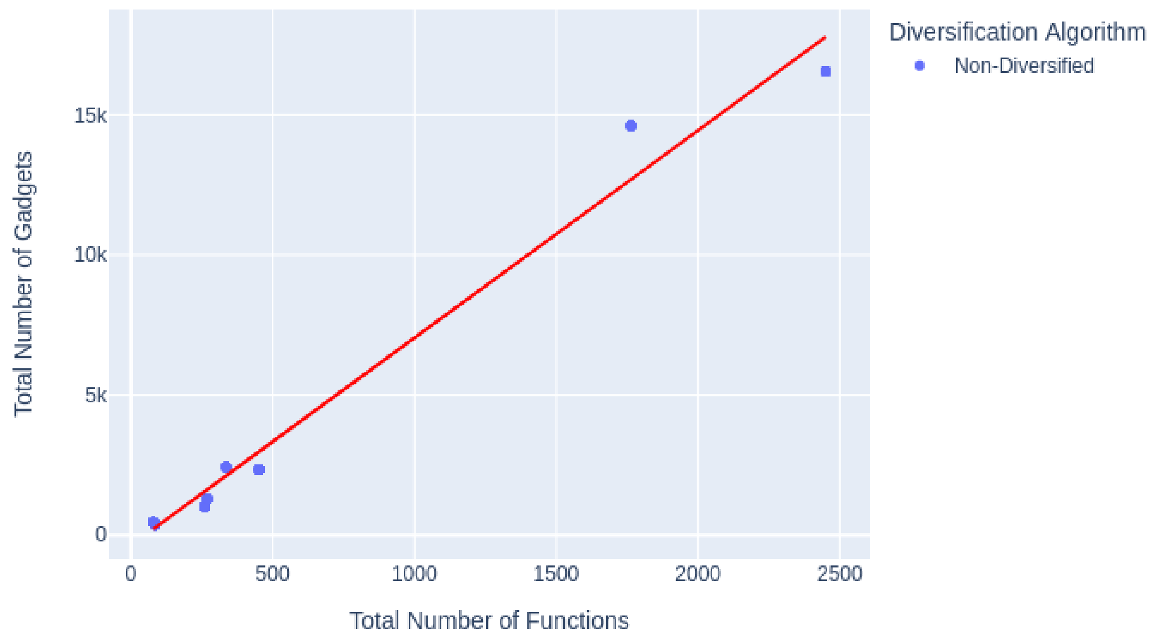

Figure 9: Function and Gadget Relationship Non-Diversified: Real-World

Following the analysis of the original binaries, the same relationship was plotted with diversified variants. This plotting was done to compare results with those originally found in [51]. Authors in that work noticed that when any diversification algorithm is applied, there is an increase in the total number of gadgets.

Figure 10**Error! Reference source not found.** shows the same relationship, the number of functions and gadgets. However, in this figure, apart from the original binaries (red diamonds), programs compiled with all transformations (sub bcf fla) are also shown (blue circles).

The results in this figure showed something that was not expected: diversification transformations are not only increasing the number of gadgets in a program, but they are also increasing the number of functions in a program. This anomaly hints that during compilation, to

include the transformation of the diversification engine, the compiler adds pre-made functions that have some of the changes already implemented.

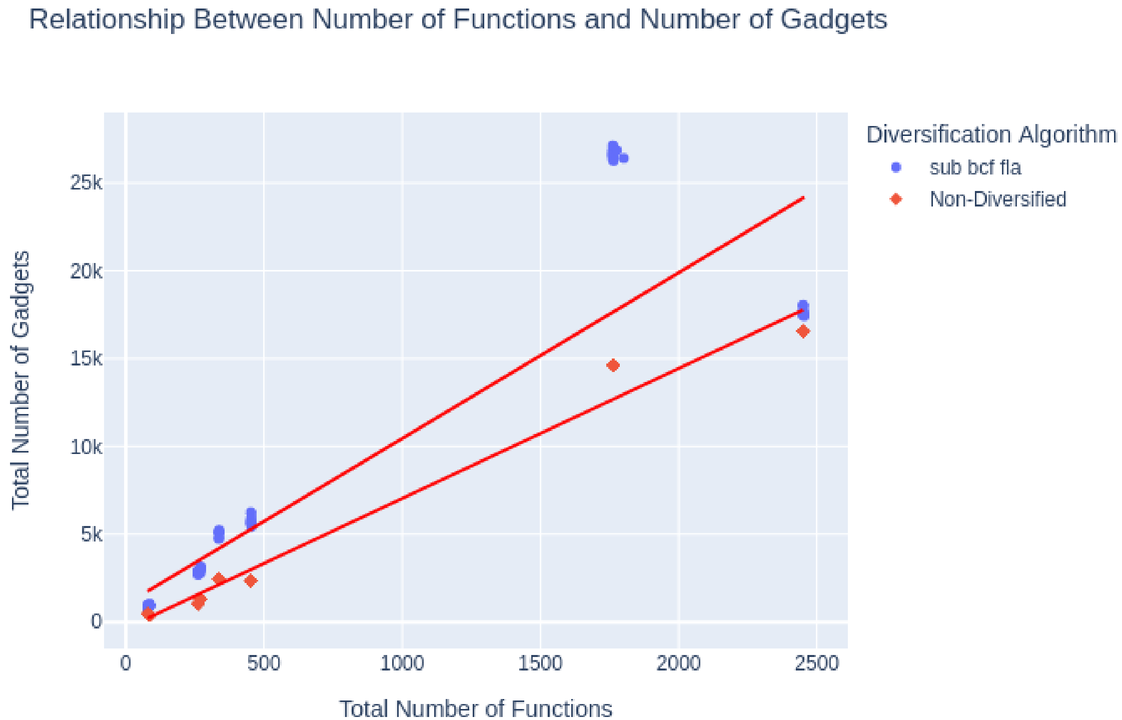

Figure 10: Function and Gadget Relationship All Algorithms and Non-Diversified: Real-World

Another anomaly identified, shown in the figure above, is an apparent increase in the number of gadgets. This increase is not as significant as was expected.

As part of the preliminary analysis for this work, the GNU Coreutils dataset was compiled in the same manner explained in chapter 5. However, a much larger increase in both gadget count and function count was observed in the Coreutils dataset. Figure 11Figure 11 shows the relationship between the number of functions and the number of gadgets for the Coreutil dataset. Both the original and binaries diversified are displayed with all transformations applied.

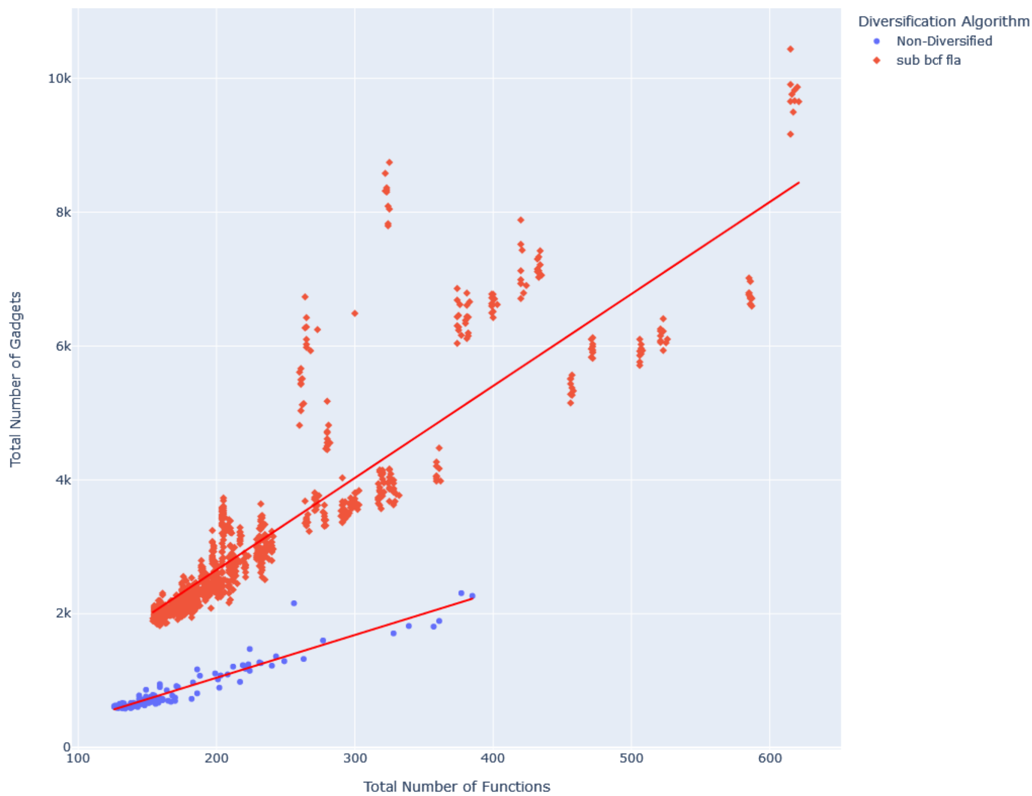

Figure 11: Function and Gadget Relationship All Algorithms and Non-Diversified: Coreutils

As seen in the figure above, Coreutils has only one similarity with the real-world dataset: an increase in the number of functions and gadgets. This inconsistency leads to the conclusion that the diversification engine affects binaries in unpredictable ways, which may be caused by the random seed used in Obfuscator-LLVM.

6.1.2 Surviving Gadgets

The second phase of this work was to understand how diversification affects the total number of surviving gadgets between the original binaries and their respective diversified variants. For this analysis, surviving gadgets are defined as the same gadget found in the same memory

address. Surviving gadgets were of particular interest because while the total number of similar gadgets would help an attacker, in theory, the attacker would need and additional vulnerability to leak the new location of the gadgets. Whereas with surviving gadgets, a new vulnerability would not be needed as attacker can easily modify their exploit without much difficulty. If the total number of surviving gadgets is large enough, the probability that they might be helpful to an attacker grows. Furthermore, suppose an attacker is able to identify these gadgets. In that case, they might be able to develop a successful exploit that can be used across all binaries, even if diversification is applied.

For this work, two areas were analyzed. First, I calculated the average percentage of surviving gadgets between the original non-diversified binary and all of the variants. Second, the shared gadgets were extracted and analyzed to understand if the surviving gadgets were substantial enough for attackers to generate a meaningful exploit that they could re-use.

To identify the surviving gadgets. ROPPER was used to find all the gadgets and addresses for each binary. Afterward, the intersection was calculated for the variants and the original. This was used to calculate the percentage of shared gadgets. Table 1 shows the results of this analysis.

Table 1: Average Percentage of Shared Gadgets between Variants and Non-Diversified Binaries

Binary Name	**SUB**	**BCF**	**FLA**		**SUB & BCF**	**SUB & FLA**	**BCF & FLA**	**ALL**
Crossfire	2.76%	0.0%	2.72%		0.0%	2.74%	0.0%	0.0%
DNSTracker	7.14%	7.11%	7.14%		7.03%	7.14%	6.86%	6.86%
LamaHub	2.52%	2.44%	2.33%		2.13%	2.23%	2.33%	2.42%
Mcrypt	6.00%	2.66%	2.69%		2.68%	2.71%	2.57%	2.45%
MP3Info	14.38%	7.32%	7.35%		7.15%	7.35%	6.85%	6.88%
NetPerf	2.67%	1.13%	0.98%		1.03%	1.03%	0.98%	1.14%

PDFResserect	8.79%	8.50%	8.46%		8.14%	8.46%	8.11%	8.14%
Sipp	0.22%	0.17%	0.20%		0.16%	0.21%	0.15%	0.15%
yTree	1.4%	1.41%	1.35%		1.45%	1.41%	1.46%	1.41%

As shown in Table 1, when the average percentage of surviving gadgets is analyzed, it is clear that while diversification does add new gadgets to the binary, it is still highly effective because it moves gadgets around the address space. This movement makes it harder for an attacker to take advantage of all the gadgets in the binary. These results are similar to the ones observed in [12]. However, the significant difference is that in [12], the authors did not analyze the type of gadgets that survive diversification to understand if an attacker can create an exploit with them. Therefore, I then decided to extract and analyze those survivor gadgets.

When analyzing the gadgets that survive diversification, I discovered that in most cases, the ones that survive diversification are actually not enough for an attacker to generate a meaningful exploit. Figure 12: Surviving Gadgets between Original and Variants PDFResserectFigure 12 shows an example of the surviving gadgets found in all variants for PDFResserect, the binary with the highest percentage of shared gadgets overall.

```
je 0xea2; call rax; ',
ret; ',
test rax, rax; je 0xea2; call rax; ',
add byte ptr [rax], al; ret; ',
ret 0x8b48; ',
pop rbp; ret; ',
add byte ptr [rax], al; add byte ptr [rax], al; ret; ',
add esp, 8; ret; ',
add rsp, 8; ret; ',
add byte ptr [rax], al; add byte ptr [rax], al; add byte ptr [rax], al; ret; ',
nop dword ptr [rax]; pop rbp; ret; ',
add byte ptr [rax], r8b; pop rbp; ret; ',
add bl, dh; ret; ',
call rax; add rsp, 8; ret; ',
add byte ptr [rax], al; pop rbp; ret; ',
je 0xea2; call rax; add rsp, 8; ret; ',
test rax, rax; je 0xea2; call rax; add rsp, 8; ret; ',
add byte ptr [rax], al; add byte ptr [rax], al; add byte ptr [rax], al; add byte ptr [rax], al; ret; ',
nop dword ptr [rax + rax]; pop rbp; ret; ',
add byte ptr [rax - 0x7b], cl; sal byte ptr [rdx + rax - 1], 0xd0; add rsp, 8; ret; ',
add byte ptr [rcx], al; pop rbp; ret; ',
and byte ptr [rax], al; test rax, rax; je 0xea2; call rax; ',
sal byte ptr [rdx + rax - 1], 0xd0; add rsp, 8; ret; ',
test eax, eax; je 0xea2; call rax; add rsp, 8; ret; ',
jmp rax; ',
test eax, eax; je 0xea2; call rax; ',
call rax; '
```

Figure 12: Surviving Gadgets between Original and Variants PDFResserect

As seen in the figure above, while some gadgets might seem problematic, such as “call rax” and “jmp rax.” These gadgets, hi-lighted in yellow in Figure 12, only allow an attacker to launch a Denial of Service (DOS) attack. This problem also carries over to scenarios where potentially dangerous gadgets such as *syscall* are shared between variants. When analyzing Crossfire-Server, I found potentially hazardous gadgets that were shared. I do not find this particularly worrying, as apart from a few *syscall* gadgets, no other gadgets that could modify registers were shared between variants. Furthermore, as shown in Table 1, the overall percentage of shared gadgets changes dramatically between all variants, further demonstrating the strength of diversification and its ability to reduce the number of successful exploit generation. In this analysis, there did not appear to be enough surviving gadgets to create more advanced exploits. This appearance carries over to larger binaries, such as the OpenSSL program in the dataset used in this work.

6.1.3 Exploit Generation

The final phase of this work was to analyze how different diversification algorithms affect exploit generation. Using ROPPER's exploit generation functionality, target binaries were scanned and two separate ROP chains were created. First, *execve* ROP chains for all binaries in our data set were created, followed by *mprotect* ROP chains. In this section, I will present the results from the analysis done on a program in our dataset, Sipp, and report on our observations on how different diversification algorithms affect the resulting chain. I decided to present Sipp for showing the *execve* analysis while selecting binaries for the dataset. It was discovered that Sipp was the only binary that had a tool-generated *execve* ROP chain exploit in ExploitDB. Therefore, because it was known that Sipp had all of the instructions needed to generate a ROP chain, it was the perfect candidate to demonstrate the changes in exploits.

```
# 0x0000000000485698: pop rax; ret;

# 0x000000000043473a: pop rdx; ret;

# 0x000000000043f8fd: mov qword ptr [rdx], rax; nop; leave; ret;
# 0x0000000000485698: pop rax; ret;

# 0x000000000043473a: pop rdx; ret;

# 0x000000000043f8fd: mov qword ptr [rdx], rax; nop; leave; ret;
# 0x0000000000458933: pop rdi; ret;

# 0x0000000000458931: pop rsi; pop r15; ret;


# 0x000000000043473a: pop rdx; ret;

# 0x0000000000485698: pop rax; ret;

# 0x000000000042381d: syscall;
```

Figure 13: ROPPER Generated execve for Sipp

Figure 13 shows the ROPPER-generated ROP chain. My results show interesting results when analyzing the generated exploits for the diversified variants. When the Instruction Substitution algorithm was applied, ROPPER could develop a partial ROP chain for 80% of

variants. This work defines a partial ROP chain as one where one or more critical gadgets are missing. In the remaining 20% of variants, ROPPER could not generate a ROP chain, as it could not identify the necessary gadgets needed to create a ROP chain. These results are identical when the Bogus Control Flow algorithm is applied.

When looking at the generated exploit chain for the Control Flow Flattening algorithms, I could see that ROPPER found enough gadgets to create a chain for only one variant. However, this chain is also a partial ROP chain, as the *syscall* gadget was missing. With this algorithm, only one binary had a complete exploit chain with a *syscall* instruction. Outside of these two exploit chains for the remaining 80% of variants, ROPPER could not find any valuable gadgets to generate a ROP chain.

When I combine algorithms, there are minor changes to the results observed. When variants are compiled using the Bogus Control Flow and Control Flow Flattening, 40% of our variants have full ROP chains with a *syscall* instruction, 50% have a partial ROP chain, and 10% of variants, a ROP chain was not generated.

When Instruction Substitution and Control Flow Flattening algorithms are combined, only 20% of variants had a complete ROP chain, 40% had no ROP chain generated, 30% had a partial ROP chain with the missing *syscall*, and only 10% had no ROP chain but did have a *syscall* gadget available.

When binaries are compiled using the Instruction Substitution and Bogus Control Flow, 60% had a partial ROP chain (missing *syscall*), 20% had an entire ROP chain, 10% had no ROP chain, and 10% only had a *syscall* gadget available.

Finally, when all diversification algorithms are used during compilation, 30% had full ROP chains, 60% had partial ROP chains, and 30% had no ROP chains.

Following the generation and analysis of the *execve* exploit chain, exploits were generated for another famous exploit, *mprotect*. Similar to the *execve* analysis, exploits were developed for all binaries in our dataset. In my study, I discovered a bug in ROPPER. Because of this bug, there were minor issues with ROPPER finding the *syscall* gadget within the target binary. This bug has since been patched.

Figure 14 shows an example of a generated *mprotect* ROP chain for the non-diversified version of Crossfire-Server.

```
# 0x00000000004b417c: pop rdi; ret;

# 0x00000000004b3ee9: pop rsi; ret;

# 0x00000000004d9675: pop rdx; ret;

# 0x00000000004b9967: pop rax; ret;
```

Figure 14: ROPPER Generated mprotect Exploit for Crossfire-Server

The results for *mprotect*, surprisingly, are different than those for *execve*. I noticed that in 90% of generated exploits, there exist enough gadgets in the diversified variants such that an attacker can create a similar exploit to the original even when all diversification algorithms are applied. An example is shown in Figure 15.

```
# 0x00000000007ea718: pop rdi; ret;

# 0x00000000004654b4: pop rsi; ret;

# 0x00000000005ce6ce: pop rdx; ret;

# 0x000000000053ccec: pop rax; ret;
```

Figure 15: ROPPER Generated mprotect Exploit for Crossfire-Server All Diversification Algorithms Applied

As seen in the Figure above, there is very little difference between the diversified variant and the original binaries. This observation is especially surprising after analyzing the results from the *execve* generated exploit. This led me to hypothesize that for an *execve* exploit to be effective, it requires a gadget that falls into the Load/Store criteria. This gadget type is often called a *'write-what-where'* gadget. These gadgets are used to allow an attacker to write the string *'/bin/sh'* to a writable section of memory within the binary, usually in the *.bss* section of memory. The attacker would then load the address where that string is stored to a register. An example of this gadget would be: **"mov qword ptr [rdx], rax; ret;."** These sets of gadgets appear to be rare in smaller programs and not found in abundance, whether or not diversification is applied. However, in other exploit types such as *mprotect*, these *write-what-where* gadgets are not required but require a different limited set of gadgets. This set of gadgets is limited to gadgets that only need immediate loading values to the registers: **RDI**, **RSI**, **RDX R10**, **R8**, **R9**, and **RAX,** which are the first six arguments in Linux systems for a function.

These register-populating gadgets are abundant in binaries due to restoring registers to a previous state before a function is called, and therefore might explain why diversification does not eliminate these gadgets. In larger binaries, however, the analysis indicates this explanation is not the case. In my research with OpenSSL, I noticed that ROPPER successfully created both a full *execve* and *mprotect* exploit. This is primarily due to how large these programs are; as such, ROPPER has more gadgets to select from and can select from areas that might not be affected by diversification.

This work is just a small step in analyzing software diversification algorithms' impact on binaries. In this section, I presented our results by looking at two exploits. I can begin to gain

insight into which gadgets can be diversified away and which cannot because they are critical to the binary or returning a binary to a previous state.

6.2 QUANTIFICATION OF THE IMPACT OF SOFTWARE DIVERSIFICATION

6.2.1 Diversification on Attack Resistance

After diversifying all of the binaries in the dataset, the results generated when calculating the Attack Resistant scores, refer to section 5.2.2 for details on how the Attack Resistance score was calculated, show that, in most cases, Instruction Substitution is the best algorithm for Attack Resistant. To offer randomization while reducing the total number of ROP gadgets added. Table 2 details the results for several binaries in the dataset. Ideally, an Attack Resistance score that is as close to the Attack Resistance score of the original binary is desired as that means the number of ROP gadgets is not being incremented. Whereas a lower score indicated that diversification is increasing the total amount of ROP gadgets in the compiled variant. Results for the full dataset can be found in Appendix I.

Table 2: Attack Resistance Score Results
(Refer to Section 5.2.2 for Calculation Details)

Binary Name	Original	SUB	FLA	BCF	SUB & BCF	SUB & FLA	BCF & FLA	ALL
Base64	0	-0.22 – 0.34	-1.93 - 2.18	-7.34 - 8.78	-10.32 - 11.38	-2.07 - 2.44	-7.64 - 8.46	-15.22 - 16.37

ls	0	-0.12 - 0.20	-0.70 - 0.80	-5.39 - 5.78	-7.35 - 8.36	-0.89 - 1.03	-5.10 - 5.58	-10.31 - 11.09
CP	0	-0.16 – 0.23	-0.75 - 0.87	-5.73 - 6.21	-8.16 - 9.02	-0.92 - 1.06	-5.47 - 6.09	-11.10 - 11.92
Sha512Sum	0	-0.17 – 0.41	-0.30 - 0.39	-2.96 - 3.55	-5.42 - 6.32	-0.94 - 1.09	-3.14 - 3.74	-7.36 - 8.11
OpenSSL	0	-0.18 – 0.19	-0.31 - 0.34	-0.31 - 0.33	-2.14 - 2.19	-0.45 - 0.47	-1.10 - 1.16	-2.02 - 2.18

One observation from the table above is how much the Attack Resistance score changes with different diversification algorithms. For instance, regarding the Attack Resistance score for Base64, Table 2 details how any algorithm other than Instruction Substitution would negatively affect the Attack Resistant score; as mentioned earlier, the goal would be to have a low Attack Resistant score. This impact, however, does not appear to scale the same way for every binary. For instance, when diversifying Base64 with the Control Flow Flattening algorithm, there is a significant increase in the Attack Resistant score, as it jumps from about 1.2 to almost 3.0. Whereas ls only has a moderate jump from 1.12 to 1.71. This difference is more noticeable when compared against a larger binary such as OpenSSL. For instance, when the binaries in this dataset are compiled with all diversification algorithms, the Attack Resistance Score increases significantly. However, with OpenSSL this increase, while still significant, is not to the scale of the other binaries. This increase may be due to the large size of OpenSSL. For instance, since OpenSSL has plenty of ROP gadgets, any change in ROP gadgets might not be as impactful as smaller binaries with fewer ROP gadgets.

6.2.2 Diversification on Exploit Complexity

When analyzing the dataset's results for the Exploit Complexity score, refer to section 5.2.3 for details on how the Exploit Complexity score was calculated, some results stand out. Most notably, as shown in Table 2, several diversification algorithms negatively affected the Exploit Complexity score while they did well on the Attack Resistance score. Results for the full dataset can be found in Appendix J.

Table 3: Exploit Complexity Score Results
(Refer to Section 5.2.3 for Calculation Details)

Binary Name	Original	SUB	FLA	BCF	SUB & BCF	SUB & FLA	BCF & FLA	ALL
Base64	263	27 – 67	34 – 51	266 – 440	829 – 953	53 – 100	547 – 652	1565 – 1780
ls	1019	133 – 214	104 – 144	987 – 1255	2919 – 3425	196 – 277	1595 – 1789	4858 – 5343
CP	779	116 - 173	109 – 129	916 – 1119	2261 – 2694	172 – 234	1345 – 1514	3805 – 4237
Sha512Sum	296	80 – 119	35 – 55	305 – 485	968 – 1117	104 – 135	520 – 698	1632 – 1827
OpenSSL	15393	15944 - 16046	22258 - 22339	18358 – 18803	31809 – 32675	22562 – 22820	29895 – 30915	31028 – 31518

For instance, as shown in the table above, Instruction Substitution was one of the worst-performing algorithms in maximizing the Exploit Complexity score for almost all binaries, the exception being OpenSSL. These results indicate that Instruction Substitution breaks down

gadgets more, resulting in more singleton gadgets. This observation means that ROP gadgets become substantially shorter and can allow attackers to craft new exploits with little or no risk of having previously set registers clobbered or modified. As with the Attack Resistance score results, a larger binary such as OpenSSL benefits from only having Instruction Substitution applied. Again, this seems to indicate that diversification, while beneficial for all binaries, might be more effective in larger binaries. While with smaller binaries, there is an apparent trade-off.

6.2.3 Diversification on Resistance to Reverse Engineering

The final results to discuss are those of the Resistance to Reverse Engineering scores, refer to section 5.2.4 for details on how the Resistance to Reverse Engineering score was calculated,. The most surprising discovery from the results shown in Table 4 appears to be that all algorithms effectively increase the Resistance to Reverse Engineering scores. While these results were expected from algorithms such as Control Flow Flattening or Bogus Control Flow, which effectively modified the control flow structure of the compiled binary, the increase in Resistance to Reverse Engineering score is not expected from the Instruction Substitution algorithm. Results for the full dataset can be found in Appendix K.

Table 4: Resistance to Reverse Engineering Score Results
(Refer to Section 5.2.4 for Calculation Details)

Binary Name	Original	SUB	FLA	BCF	SUB & BCF	SUB & FLA	BCF & FLA	ALL
Base64	638	1536 – 1640	2323	2680 – 2856	2726 – 2838	2323	4191 – 4330	4166 – 4382

ls	2909	3305 – 4087	6272	6669 – 6849	7369 – 8058	6272	114020 – 11658	11342 – 11742
CP	1925	2563 – 3162	4750	5101 – 5293	5147 – 5382	4750	8607 – 8920	8583 – 8926
Sha512Sum	682	1312 – 1555	2298	2424 – 2594	2504 – 2566	2298	4143 – 4272	4124 – 4307
OpenSSL	30520	30549 - 30597	77792 – 77803	63545 – 64765	65164 – 66292	77738 – 77816	144340 – 145741	152999 – 155075

As shown in the table above, Instruction Substitution offers moderate Resistance to Reverse Engineering for all binaries. In some cases, Instruction Substitution can improve the Resistance to Reverse Engineering score by as much as double, as with Base 64 and Sha512sum. For larger binaries, however, such as OpenSSL, this increase is minimal but should still be noted.

6.3 ALGORITHM SELECTION CASE STUDY

To demonstrate how the system developed can recommend the best diversification algorithm and the performance impact associated with each diversification algorithm, this work presents two case studies.

6.3.1 Setup

The system developed was installed on a Dell Precision 7720 laptop with a Xenon processor and 64 gigabytes of memory. As a part of this case study, GNU Coreutils and vulnerable binaries described in chapter 4 were diversified using Obfuscator-LLVM. All the programs in this

dataset were diversified using all three diversification algorithms and their combinations ten times. All binaries had all three quantification criteria calculated using the analysis component following the diversification process. In addition, all of the binaries in this dataset had two separate performance metrics collected using Perf, CPU clock time, and total execution time.

After completing the previous step, the algorithm selector component selected the best algorithm for two different percentage combinations. Finally, the graphing engine utilized the information from the selector component and generated Perf output to generate a graph of the performance impact and display the best diversification algorithm.

6.3.2 Maximizing Attack Resistance

For the first use case scenario to demonstrate the inner working of the selector system, the target system has limited memory and hard drive space for this first scenario. This system could be either router, industrial control device, or a program running on an automobile. For this scenario, Base64 and Sha512sum were selected, as these programs are standard programs used in smaller systems such as routers. This scenario is focused on maximizing the Attack Resistance score as this score offers diversification without drastically modifying the program's structure, unlike control flow flattening and bogus control flow.

After using the developed system to create variants of both programs, calculate their respective Attack Resistance Scores, and run the performance analysis, the system generated two different graphs. Figure 16 displays the results for Base64 after using the selector system developed in this work. In this figure, the x-axis is the name of the diversification algorithms. The algorithms are color coded as follows: Instruction Substitution in blue, Control Flow Flattening in red, Bogus Control Flow in green, Instruction Substitution combined with Bogus Control Flow in

purple, Instruction Substitution combined with Control Flow Flattening in orange, Bogus Control Flow combined with Control Flow Flattening in light blue, and all diversification algorithms in pink.

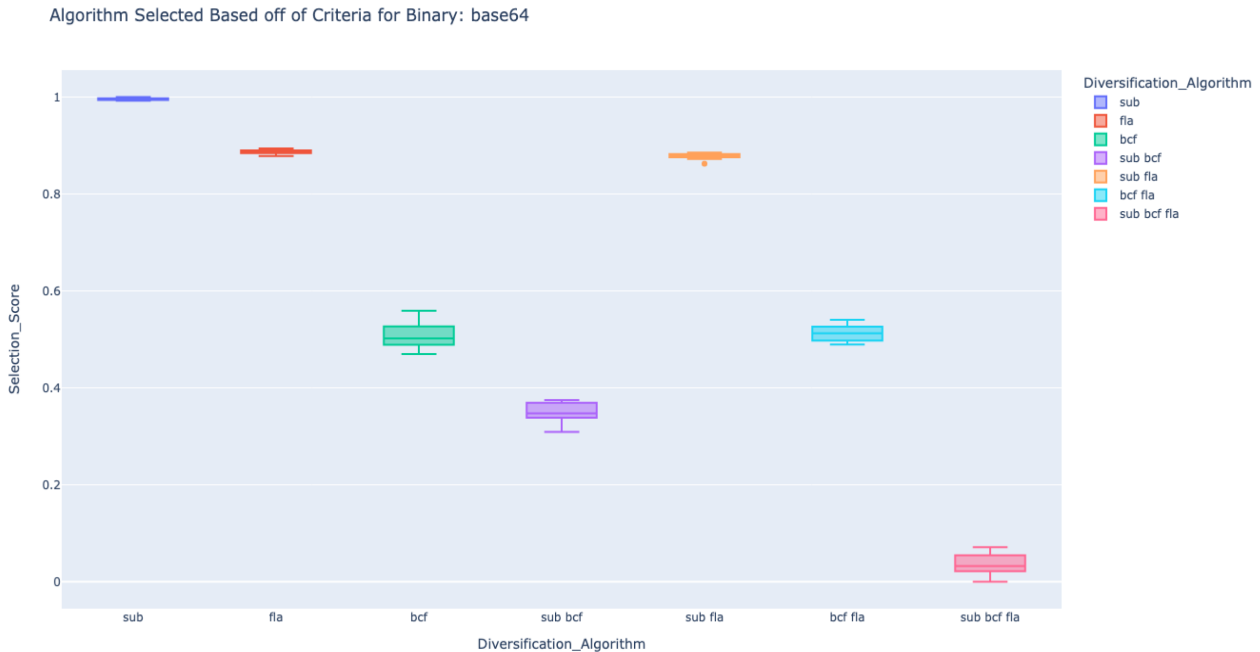

Figure 16: Base64 Best Attack Resistance Algorithm

The graph generated by the system compares algorithms and visualizes where each algorithm or combination of algorithms scores based on the user's input.

As shown in the figure above, after analyzing all variants, the algorithm selector system would recommend that Base64 binaries be compiled with the Instruction Substitution algorithm, followed by the Control Flow Flattening algorithm.

Furthermore, when the system analyzes performance metrics associated with diversified binaries, the selector system also displays a performance graph, as shown in Figure 17 detailing the performance impact associated with each binary.

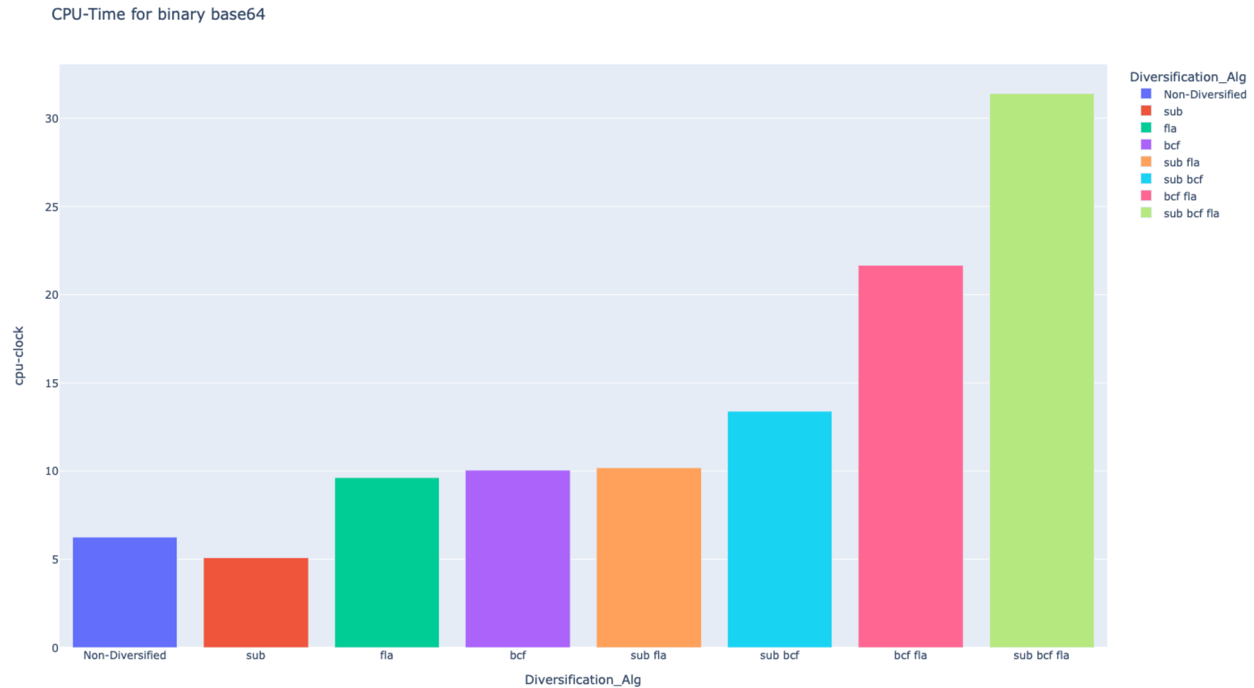

Figure 17: CPU-Time Impact for Base64

As seen in the box plot generated by this system, the results indicate that Instruction Substitution would be the best algorithm to use as it would be the algorithm that would offer the randomness of diversification while minimally impacting performance.

While the selector system recommends instruction substitution as the best algorithm for Base64, this recommendation is not necessarily the case for the second binary, Sha512Sum. When the selector system analyzes Sha512sum, the best algorithm can be Instruction Substitution or Control Flow Flattening. Figure 18 displays the complete results generated from the selector algorithm.

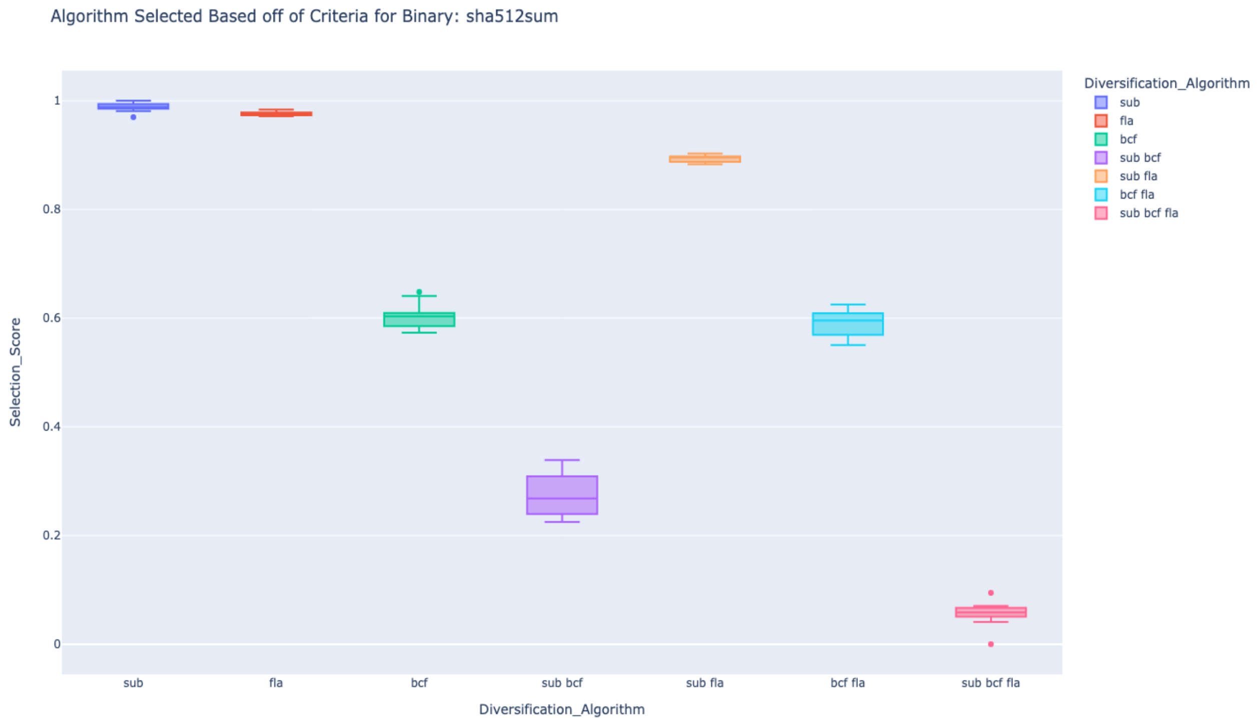

Figure 18: Sha512Sum Best Attack Resistance Focus

As seen in the figure above, both Instruction Substitution and Control Flow Flattening are potential candidates as the optimal algorithm based on the criteria the analyst wants to maximize. Although Instruction Substitution and Control Flow Flattening might be the best algorithm for maximizing Attack Resistance, the selection system details that using these two algorithms will have some performance penalty with Sha512Sum. Figure 19 displays the performance penalties associated with Sha512Sum.

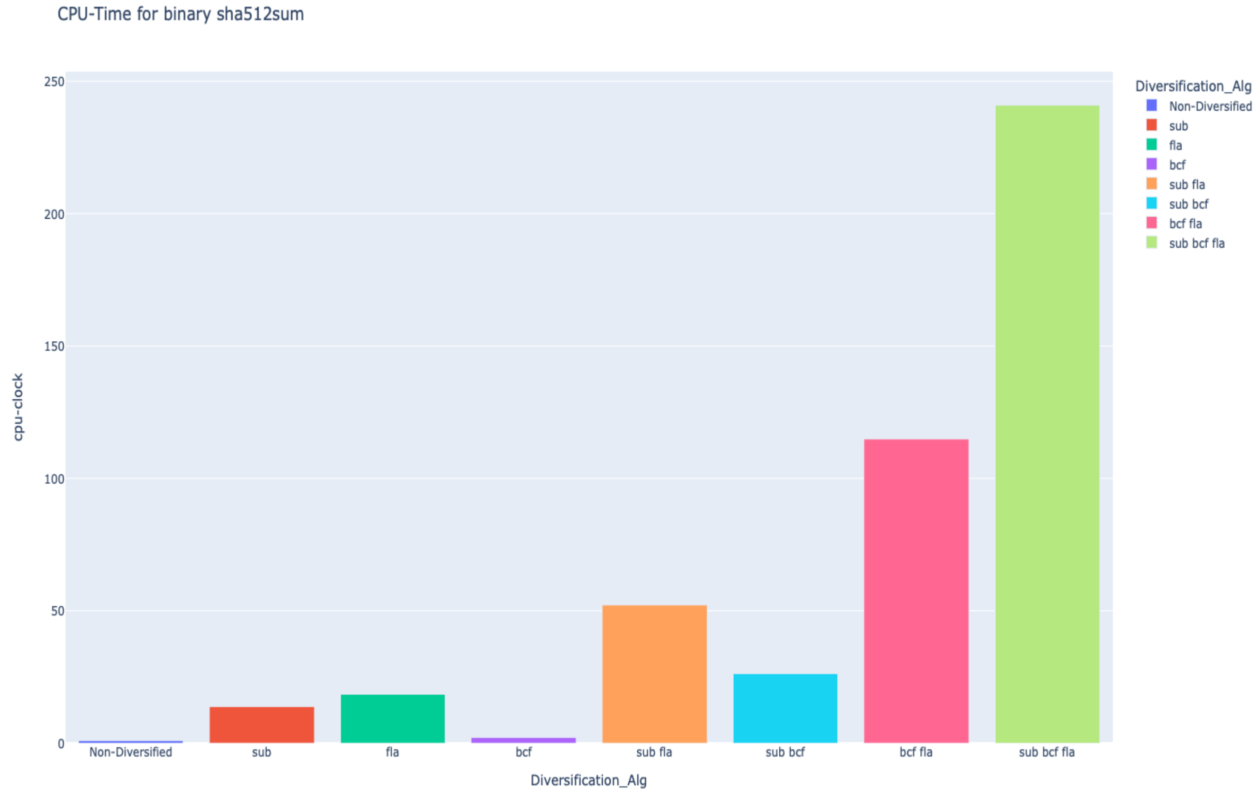

Figure 19: CPU-Time Impact for Sha512 Sum

In this scenario, the diversification algorithm that would have a minimal impact on performance would be Bogus Control Flow. This scenario demonstrates the trade-offs associated with diversification and the need to leave the final decision to the end user regarding which algorithm to use.

6.3.3 Balanced Diversification

For this final scenario, we suppose the system is your standard workspace; this system could be either an all-in-one workstation, a desktop computer, or a modern laptop. Unlike the previous scenario, this scenario will demonstrate the selector component's ability to show how diversifications can impact total execution time of the diversified binaries. Similar to the previous scenario, the binaries selected were Base64 and ls. These binaries, apart from being used in small

devices, are standard on systems running the Linux operating system. However, wherein the last scenario, the goal was to maximize the Attack Resistant score, in this scenario, the goal is to have a diversification algorithm that evenly disperses its impact. That is to say, the goal is a 33% focus on Attack Resistance Exploit Complexity and Resistance to Reverse Engineering.

After the selector system diversifies and analyzes the binaries, the system generates the box plot shown in Figure 20. Similar to the previous scenario, the x-axis is the name of the diversification algorithms. The algorithms are color coded as follows: Instruction Substitution in blue, Control Flow Flattening in red, Bogus Control Flow in green, Instruction Substitution combined with Bogus Control Flow in purple, Instruction Substitution combined with Control Flow Flattening in orange, Bogus Control Flow combined with Control Flow Flattening in light blue, and all diversification algorithms in pink.

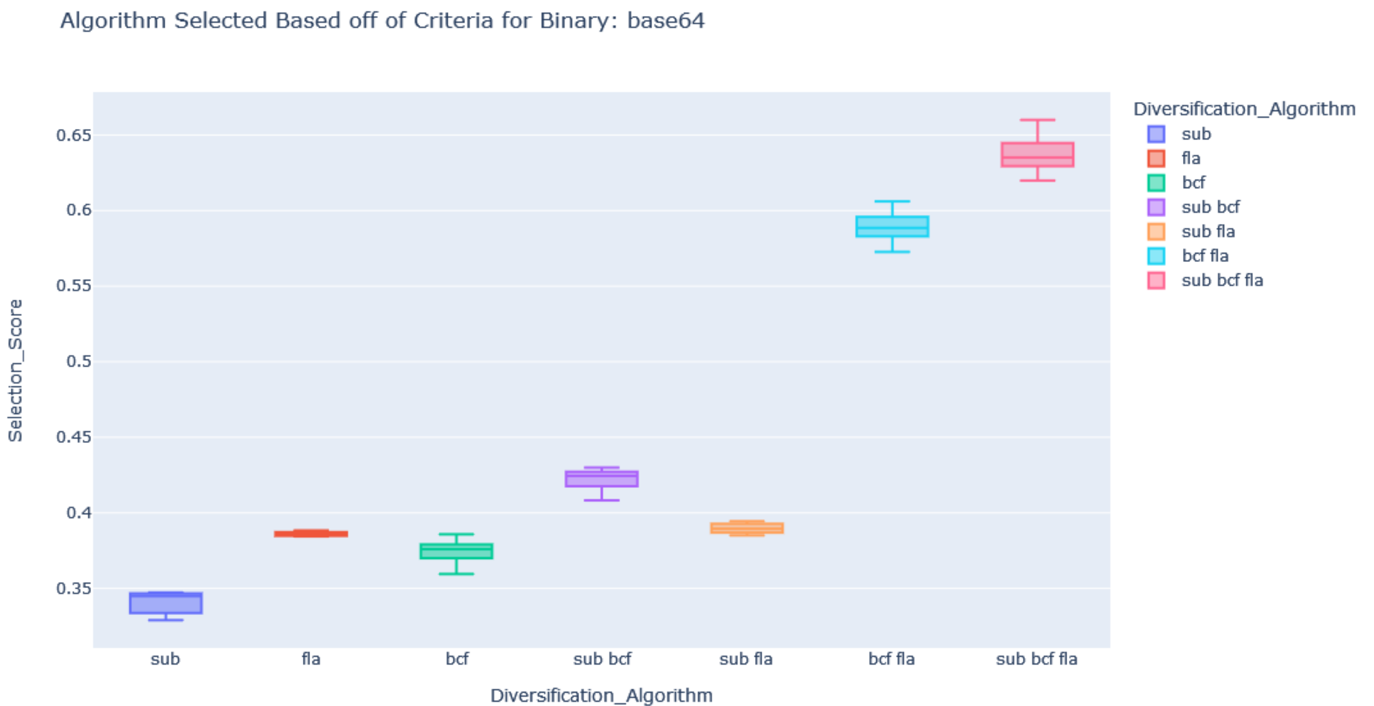

Figure 20: Base64 Best Balanced

The results from the analysis would indicate that the best diversification engine to use to balance out the impact of diversification for Base64 would be to compile Base64 with all diversification algorithms. The second best algorithm would be a binary compiled with Bogus Control Flow and Control Flow Flattening. Similar to the scenario above, the selector system also generates a graph visualizing the overall impact of diversification, except this scenario presents the total time elapsed diagram. Figure 21 illustrates the execution time associated with Base64.

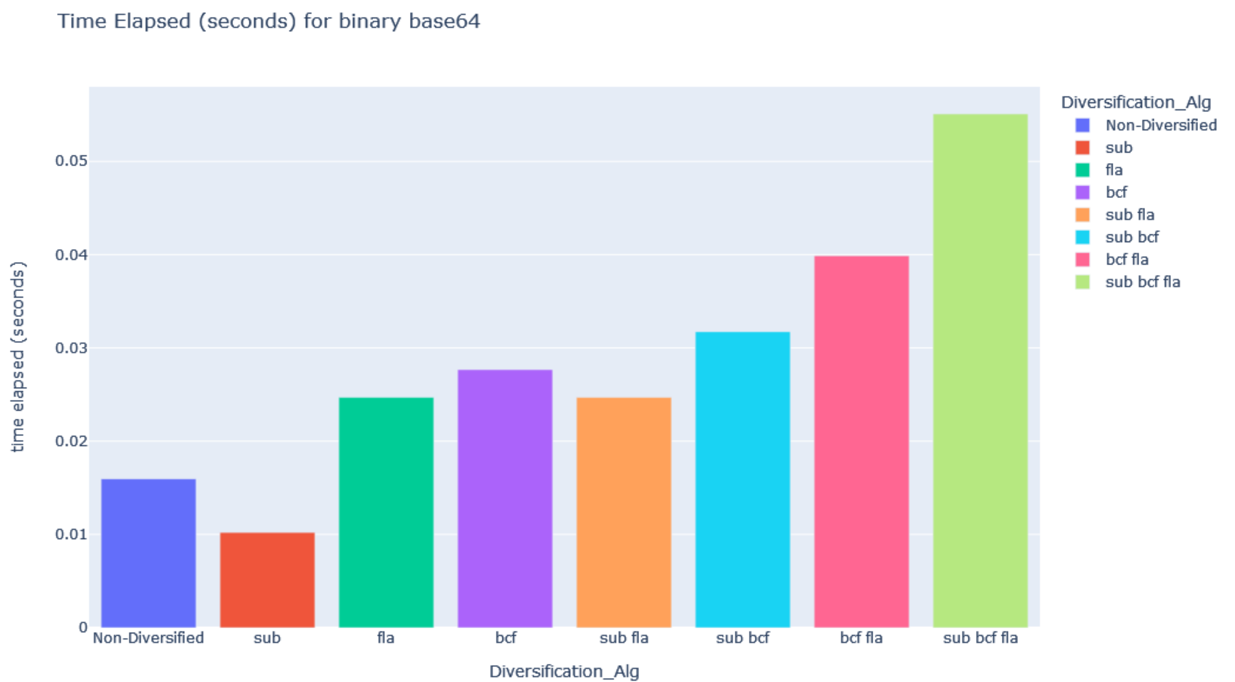

Figure 21: Base64 Total Execution Time

As shown in the figure above, diversifying with all diversification flags might be the best to offer a balanced level of diversification, but there is some performance impact. However, looking at the overall execution time, execution time only increased by .04 of a second. Because

of this minor increase, it would be appropriate to diversify using all algorithms, as the execution time impact might be negligible when running on a more extensive system.

Finally, for the second program in this scenario ls, the selector system would suggest that variants of ls be compiled using the same combination of algorithms as with Base64. Figure 22 displays the results of this analysis.

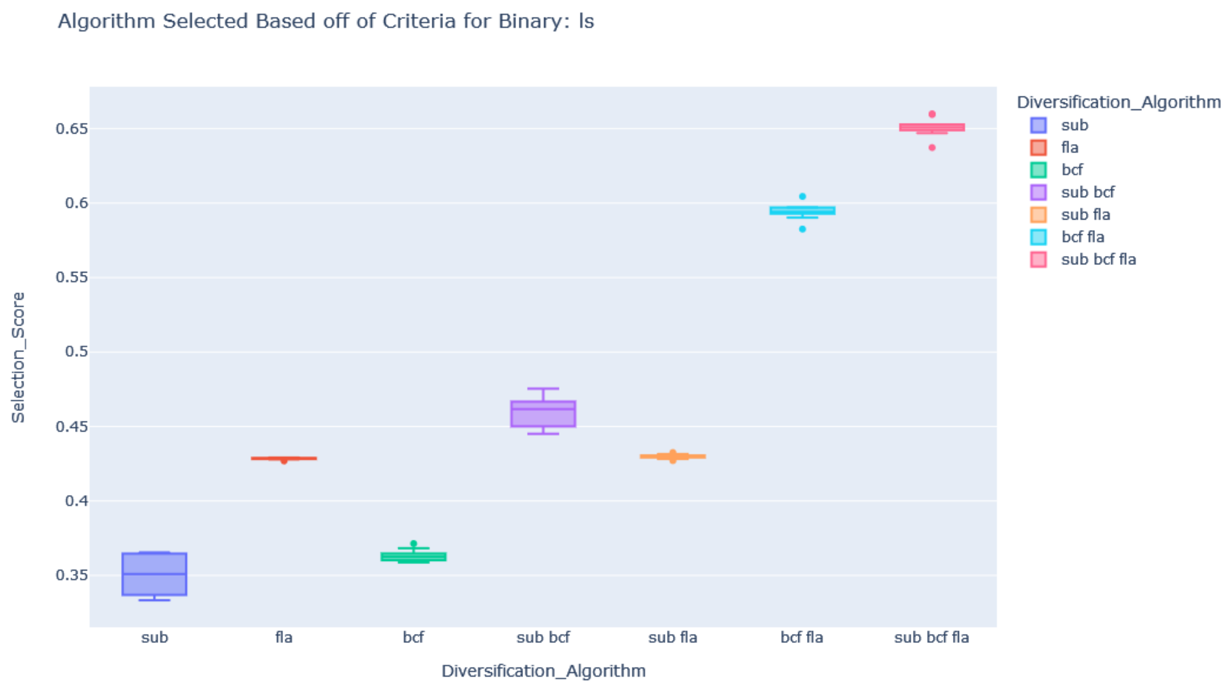

Figure 22: ls Best Balanced

One notable difference between Base64 and LS is that while Bogus Control Flow and Control Flow Flattening is the second algorithm, the impact spread is closer. This signifies that with ls, unlike Base64, there will be less variation between diversified variants. Additionally, less spread may be good if the end user wants more consistent variants.

Finally, when analyzing the total time graph similar to previous examples, compiling with all diversification flags does have a significant performance impact for ls. The result in terms of total execution time can be seen in Figure 23

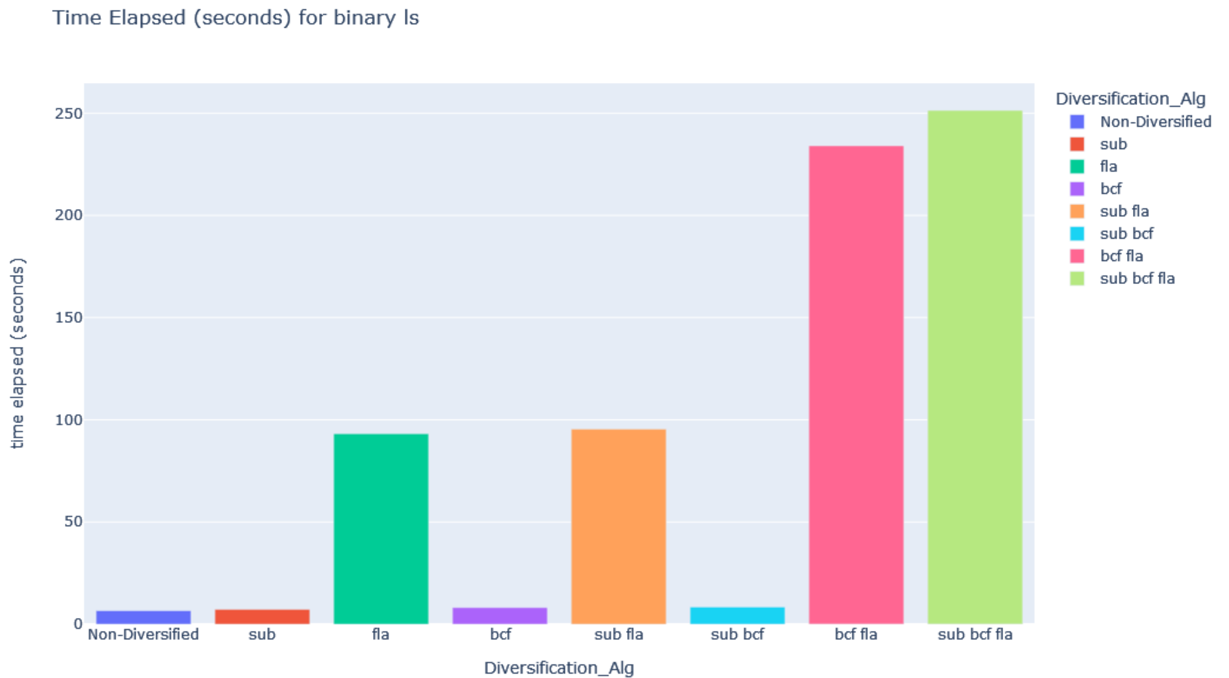

Figure 23: ls Total Execution Time

The figure shows that the total time it would take to execute using all diversification algorithms is substantially significant.

6.4 DISCUSSION

The results presented in this chapter make it clear that software diversification is highly effective at modifying the binary such that ROP gadgets move around a program's memory space. Additionally, as the results presented detail, diversification can sometimes remove critical ROP

gadgets that could affect the development of certain ROP chains, as was the case with *execve*. Moreover, this work supports theories presented in [2] and [21] in that some ROP gadgets cannot be removed through diversification. Furthermore, the results presented in this work detail that while there are ROP gadgets that are survivors between variants and the original binary, these gadgets are insufficient for an attacker to develop a significant exploit.

When discussing the impact of software diversification on binaries from a quantification aspect, it is clear that software diversification, as with everything in security, is a trade-off. While one algorithm might offer more protection in terms of the Attack Resistance score developed in this work, that algorithm might affect the Exploit Complexity score or even performance depending on the binary. This trade-off is a primary reason the selector system was developed. With the quantification metrics presented along with the selector system designed and produced, this work hopes to alleviate some of the decision-making by allowing analysts to visualize the impact.

Finally, there are several constraints to this work. The first constraint has to do with the architecture. While this work only analyzed software diversification's impact on the Intel x86-64 architecture, with development of the selector system and the quantification methods presented can easily be expanded to analyze other architectures. Another constraint associated with this work is when diversification can be introduced; in this work, diversification is introduced during the compilation phase. This means that diversification cannot be introduced on binaries that are already running on a system. Thus these systems would need to be taken offline for a short period to re-compile binaries and introduce diversification. As more diversification engines become open-sourced, this could change in the future, and diversification can be introduced without taking entire systems offline to re-compile binaries.

CHAPTER 7: CONCLUSION

7.1 SUMMARY

Although the area of software diversification is over 20 years old, most of the literature to date has primarily focused on the algorithmic development side. As a result, previous work has failed to evaluate the effects of software diversification hindering an attacker's ability to create new exploits or re-use existing exploits. Furthermore, there has yet to be an effort to quantify the impact of software diversification algorithms.

The significant contribution of this work was in gaining a greater understanding of software diversification's impact on binaries. This work analyzed software diversification's impact on binaries regarding its ability to prevent attackers from re-using their previously crafted exploit and generating a new exploit post-diversification. With the development of a case study that used real-world binaries with known vulnerabilities, this work discovered that software diversification is highly effective at preventing certain classes of exploits, such as *execve*. This work also identified that for other categories of ROP exploits, such as *mprotect*, diversification is limited to modifying the length of gadgets and moving them around the program's address space. This work is the first of its kind to examine how software diversification impacts exploit development using vulnerable programs.

Following this analysis, this research focused on quantifying software diversification's impact on binaries. Through an in-depth literature review, this work identified three criteria that the ideal diversification algorithm should address and maximize 1) Attack Resistance, 2) Exploit Complexity, and 3) Hardening against reverse engineering. Through the methods proposed to quantify software diversification's impact, researchers can understand the tradeoffs associated with

each diversification algorithm which can be expanded and applied to compile time diversification and other diversification approaches. The results observed in terms of quantifying the impact of software diversification concerning the quantification methods show that software diversification is a trade-off. While one diversification algorithm might maximize one criterion, it could also lower the score of another criterion.

To address this trade-off, this work expanded on previous work presented and developed a selector tool. This tool allows operators to streamline the analysis process and identify and visualize the best diversification algorithm for the criteria they want to maximize. Finally, similar to previous work the selector tool still allows for the performance impact analysis of diversified binaries. This selector system makes it easier for analysts to select the best diversification algorithm based on their criteria and visualizes the results to allow them to see and understand how each algorithm compares against the other. Through the use of the selector system developed in this dissertation, analysts can make informed decisions when selecting the best algorithm for their needs.

7.2 FUTURE WORK

Future work in this area can be divided into three different sections. The first section concerns the selector system. Future work in this area will expand our selector system to incorporate more diversification engines. As previously mentioned, diversification engines currently presented in the literature have been primarily closed-sourced. As a result, this work has mainly used Obfuscator-LLVM because it is open-sourced, used extensively in the literature, and used in industry. These new diversification engines do not have to be compiler-based either; as demonstrated initially, the selection system utilized a post-compilation diversification engine.

With the architecture and structure of the selection tool, integrating new diversification toolsets and analysis tools can be done quickly and easily.

Regarding the selector system section, the current selection system only records CPU time and total execution time. Another area of future work would be to record other metrics, such as CPU usage, memory usage, and power consumption. As well as creating additional components by developing a prediction component using AI and Machine Learning to predict the impact of different diversification techniques.

The second section of future work concerns the analysis portion of this dissertation. One area available for future work would be to expand the dataset used; this work envisions a consistently growing dataset to allow for a broader picture of how diversification affects binaries. Along with the growing dataset, this work anticipates introducing new quantification methods similar to those presented in the software obfuscation world, as there exists an overlap between the two. Additionally, future work could examine the impact of software diversification on binaries with multiple vulnerabilities. This work only investigated software diversification's effect on binaries with only a fundamental buffer overflow vulnerability. However, it would be interesting to examine diversification's role in preventing ROP-based exploits when combined with other vulnerabilities, such as a buffer overflow vulnerability with an information leak.

The final section of future work is related to the diversification standpoint. One area open for future work is something that this work is calling Percentage-Based Diversification. Before transformations are applied to a binary, a list of all the locations where those transformations will be used is created. Unfortunately, operators cannot select a subset of that list, making it an all-or-nothing ordeal. With Percentage-Based Diversification, operators could randomly select a subset of that list, allowing for fine-tuning of diversification. This has the potential to improve the results

observed in this work and will allow for more flexibility when diversifying binaries. The final area of future research, form the diversification standpoint, is expanding this work to different architectures. By using the methods, and the selector system developed in this work, future work can examine the impact of diversification on different architectures such as ARM. Through the use of Obfuscator-LLVM, diversifying for multiple architectures can be done with minimal complications. Additionally, because ROPPER utilizes the Capstone, Unicorn, and Keystone to disassemble binaries, the only limitation would be architectures also supported by those Python libraries.

REFERENCES

[1] J. C. Foster and J. Deckard, "Buffer Overflow Attacks: Detect, Exploit, Prevent," Rockland, Syngress Publishing, Inc, 2005, p. 403.

[2] S. Schirra, "Ropper - rop gadget finder and binary information tool," 2013. [Online]. Available: https://scoding.de/ropper/.

[3] M. Dowd, J. McDonald and J. Schuh, "The Art Of Software Security Assessment," Addison Wesley, 2006, p. 89.

[4] D. Day and Z. Zhengxu, "Protecting Against Address Space Layout Randomization (ASLR) Compromises and Return-to-Libc Attacks Using Network Intrusion Detection Systems," in *International Journal of Automation and Computing*, 2011.

[5] Z. Sergey, "VirtualBox E1000 Guest-to-Host Escape," November 2018. [Online]. Available: https://github.com/MorteNoir1/virtualbox_e1000_0day. [Accessed December 2018].

[6] D. Reyes, J. C. Acosta, A. Escobar De La Torre and S. I. Salamah, "A System for Analyzing Diversified Software Binaries," in *MilCom*, Norfolk, 2019.

[7] H. Koo, Y. Chen, L. Lu, V. P. Kemerlis and M. Polychronakis, "Compiler-Assisted Code Randomization," in *2018 IEEE Symposium on Security and Privacy (SP)*, San Jose, Ca, 2018.

[8] P. Larson, A. Homescu, S. Brunthaler and M. Franz, "SoK: Automated Software Diversity," in *IEEE Symposium on Security and Privacy*, San Jose, CA, 2014.

[9] J. Salwan, "ROPgadget -Gadgets finder and auto-roper," 12 March 2011. [Online]. Available: https://github.com/JonathanSalwan/ROPgadget. [Accessed 20 April 2020].

[10] C. Team, "mona.py – the manual," 14 July 2011. [Online]. Available: https://www.corelan.be/index.php/2011/07/14/mona-py-the-manual/. [Accessed 20 April 2020].

[11] E. J. Schwartz, T. Avgerinos and B. David, "Q: Exploit Hardening Made Easy," in *USENIX Security Symposium*, 2011.

[12] J. Coffman, D. M. Kelly, C. C. Wellons and A. S. Gearhart, "ROP Gadget Prevalence and Survival under Compiler-Based Binary Diversification Schemes," in *Proceedings of the 2016 ACM Workshop on Software PROtection*, Vienna, Austria, 2016.

[13] A. One, "Smashing The Stack For Fun And Profit," Phrack, 1996.

[14] J. v. Neumann, "The First Draft Report on the EDVAC," Moore School of Electrical Engineering University of Pennsylvania, Pennsylvania , 1945.

[15] National Institute of Standards and Technology, "National Vulnerability Databse Statistics Results," [Online]. Available: https://nvd.nist.gov/vuln/search/statistics?form_type=Basic&results_type=statistics&query=Buffer+Overflow&search_type=all&isCpeNameSearch=false. [Accessed 24 October 2022].

[16] B. Potteiger, J. Mills, D. Cohen and P. Velez, "RUCKUS: A Cybersecurity Engine for Performing Autonomous Cyber-Physical System Vulnerability Discovery at Scale," in *Proceedings of the 7th Symposium on Hot Topics in the Science of Security*, New York, NY, USA, 2020.

[17] E. Buchanan, R. Roemer and S. Savage, *Return-Oriented Programming: Exploits Without Code Injection,* Las Vegas, 2008.

[18] R. Romer, E. Buchanan, H. Shacham and S. Savage, "Return-Oriented Programming: Systems, Languages, and Applications," *ACM Trans. Inf. Syst. Secur.,* vol. 15, no. 1, 2012.

[19] A. Bansal and D. Mishra, "A practical analysis of ROP attacks," *CoRR,* vol. abs/2111.03537, 2021.

[20] P. Larsen, S. Brunthaler and M. Franz, "Automatic Software Diversity," 2015.

[21] C. Liming and A. Algirdas, "N-version Programming: A Fault Tolerance Approach to Reliability of Software Operation," in *Annual International Conference on Fault-Tolerant Computing*, 1978.

[22] V. Bharathi, "N-Version programming method of Software Fault Tolerance: A Critical Review," in *National Conference On Nonlinear Systems & Dynamics*, 2003.

[23] GCC, "Hardware Models and Configurations," [Online]. Available: https://gcc.gnu.org/onlinedocs/gcc-4.9.4/gcc/Submodel-Options.html. [Accessed 12 2 2019].

[24] T. J. S. C. S. B. P. L. a. M. F. A. Homescu, "Large-Scale Automated Software Diversity—Program Evolution Redux," *IEEE Transactions on Dependable and Secure Computing,* vol. 14, no. 2, pp. 158-171, 2017.

[25] M. Kolsek, "0patch Blog," [Online]. Available: https://blog.0patch.com/2017/11/did-microsoft-just-manually-patch-their.html.

[26] M. Prasad, "Disassembly Challenges," 04 05 2003. [Online]. Available: https://static.usenix.org/event/usenix03/tech/full_papers/prasad/prasad_html/node5.html. [Accessed 9 2 2020].

[27] V. Pappas, M. Polychronakis and A. Keromytis, "Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization," in *Proceedings - IEEE Symposium on Security and Privacy*, 10.1109.

[28] "Adobe CoolType - SING Table 'uniqueName' Local Stack Buffer Overflow," [Online]. Available: http:// www.exploit- db. com/exploits/ 16619/. [Accessed 9 2 2020].

[29] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall and J. W. Davidson, "ILR: Where'd My Gadgets Go?," in *IEEE Symposium on Security and Privacy*, 2012.

[30] N. V. Database, "CVE-2006-3459 Detail," 02 08 2006. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2006-3459. [Accessed 9 2 2020].

[31] A. Gupta, S. Kerr, M. S. Kirkpatrick and E. Bertino, "Marlin: A fine grained randomization approach to defend against ROP attacks," in *NSS 2013: Network and System Security*, Madrid, 2013.

[32] "Paradyn Pro ject: UNSTRIP," 2011. [Online]. Available: http://paradyn.org/html/tools/unstrip.html. [Accessed 10 2 2020].

[33] R. Wartell, V. Moha, K. W. Hamlen and Z. Lin, "Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code," in *CCS'12: the ACM Conference on Computer and Communications Security*, Raleigh, 2012.

[34] L. Davi, A. Dmitrienko, S. Nürnberger and A.-R. Sadeghi, "Gadge Me If You Can Secure and Efficient Ad-hoc Instruction-Level Randomization," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, Hangzhou China, 2013.

[35] R. Wartell, V. Mohan, K. W. Hamlen and Z. Lin, "Binary Stirring: Self-randomizing Instruction Addresses of Legacy x86 Binary Code," in *Proceedings of the 2012 ACM conference on Computer and communications security*, Raleigh, 2012.

[36] S. Hosseinzadeh, S. Hyrynsalmi and V. Leppänen, "Obfuscation and diversification for securing the internet of things (IoT)," in *Internet of Things Principles and Paradigms*, 2016, pp. 259-274.

[37] E. Hjelmvik and W. John, "Breaking and Improving Protocol Obfuscation," Technical Report No. 2010-05.

[38] A. Homescru, S. Neisius, P. Larson, S. Brunthaler and M. Franz, "Profile-guided automated software diversity," in *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization*, 2013.

[39] D. M. Kelly, C. C. Wellons, J. Coffman and A. S. Gearhart, "Automatically Validating the Effectiveness of Software Diversity Schemes," in *2019 IEEE/IFIP International Conference on Dependable Systems and Networks Supplemental*.

[40] P. Junod, J. Rinaldini, J. Wehrli and J. Michielin, "Obfuscator-LLVM — Software Protection for the Masses," in *Proceedings of the 1st International Workshop on Software Protection,*, 2015.

[41] J. Coffman, A. Chakravarty, J. A. Russo and A. S. Gearhart, "Quantifying the Effectiveness of Software Diversity Using Near-Duplicate Detection Algorithms," in *Proceedings of the 5th ACM Workshop on Moving Target Defense*, Toronto, Canada, 2018.

[42] S. Schleimer, D. S. Wilkerson and A. Aiken, "Winnowing: Local Algorithms for Document Fingerprinting," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, San Diego, 2003.

[43] M. Henzinger, "Finding Near-Duplicate Web Pages: A Large-Scale Evaluation of Algorithms," in *Proceedings of the 29th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Seattle, 2006.

[44] H. Joshi, A. Dhanasekaran and R. Dutta, "Trading Off a Vulnerability: Does Software Obfuscation Increase the Risk of ROP Attacks," *Journal of Cyber Security and Mobility,* vol. 4, pp. 305-324, 2016.

[45] H. Shacham, "The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86)," in *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security*, 2007.

[46] S. Schirra, "Ropper - rop gadget finder and binary information tool," 2013. [Online]. Available: https://scoding.de/ropper/. [Accessed 24 07 2022].

[47] S. Wang, P. Wang and D. Wu, "Composite Software Diversification," in *IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Shanghai, 2017.

[48] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel and G. Vigna, "SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis," in *IEEE Symposium on Security and Privacy*, 2016.

[49] A. Daroc, "Exploiting More Binaries by Using Planning to Assemble ROP Attacks,," 2020.

[50] A. Nurmukhametov, A. Vishnyakov, V. Logunova and S. Kurmangaleev, "MAJORCA: Multi-Architecture JOP and ROP Chain Assembler," *Ivannikov ISPRAS Open Conference (ISPRAS),* no. IEEE, pp. 37-46, 2021.

[51] M. Research, *Z3 Theorem Prover,* Microsoft Corporation, 2012.

[52] C. Collberg, C. Thomborson and D. Low, "A Taxonomy of Obfuscating Transfomrations," The University of Auckland, 1997.

[53] S. A. Ebad, A. A. Darem and J. H. Abawajy, "Measuring Software Obfuscation Quality- A Systematic Literature Review," in *IEEE Access*, 2021.

[54] A. Calleja, J. Tapiador and C. JUan, "A Look into 30 Years of Malware Development from a Software Metrics Perspective," in *19th International Symposium on Research in Attacks, Intrusions and Defenses*, 2016.

[55] A. Calleja, J. Tapiador and J. Caballero, "The MalSource Dataset: Quantifying Complexity and Code Reuse in Malware Development,," *IEEE Transactions on Information Forensics and Security,* vol. 14, no. 12, pp. 3175-3190, 2019.

[56] NSA, "NSA/CSS," 5 March 2019. [Online]. Available: https://www.nsa.gov/Press-Room/News-Highlights/Article/Article/1775584/ghidra-the-software-reverse-engineering-tool-youve-been-waiting-for-is-here/. [Accessed 27 October 2022].

[57] V. 35, "Binary Ninja," [Online]. Available: https://binary.ninja/. [Accessed 27 October 2022].

[58] Hex-Rays, "IDA Pro," [Online]. Available: https://hex-rays.com/IDA-pro/. [Accessed 27 October 2022].

[59] Plotly, "Plotly python open source graphing library," [Online]. Available: https://plotly.com/python/. [Accessed 28 October 2022].

[60] P. H. Joshi, D. Aravindhan and D. Rudra, "Impact of Software Obfuscation on Susceptibility to Return-Oriented Programming Attacks," in *36th IEEE Sarnoff Symposium*, 2015.

[61] A. Avizienis, The Methodology of N-Version Programming, 1995.

[62] O. Foundation, "OWASP Top Ten," OWASP, [Online]. Available: https://owasp.org/www-project-top-ten/. [Accessed 17 May 2021].

[63] R. Roemer, E. Buchanan, H. Shacham and S. Savage, "Return-Oriented Programming: Systems, Languages, and Applications," *ACM Transactions on Information and System Security,* vol. 15, no. 1, 2012.

[64] B. Thomas, "U.S. Cyber Command Technical Challenge Problems Guidance," 12 March 2019. [Online]. Available: https://www.cybercom.mil/Portals/56/Documents/Technical%20Outreach/Technical%20Challenge%20Problems.pdf?ver=2019-07-02-151118-497. [Accessed 21 January 2020].

[65] M. Smithson, K. Elwazeer, K. Anand, A. Kotha and R. Barua, "Static Binary Rewriting without Supplemental Information Overcoming the Tradeoff between Coverage and Correctness," in *20th Working Conference on Reverse Engineering*, 2013.

[66] H. Ralf, H. Thorsten and F. C. Felix, "Return-Oriented Rootkits: Bypassing Kernel Code Integrity Protection Mechanisms," in *18th USENIX Security Symposium*, Montreal, 2009.

[67] P. Larsen, A. Homescu, S. Brunthaler and M. Franz, "SoK: Automated Software Diversity," in *Proceedings of the 2014 IEEE Symposium on Security and Privacy*, 2014.

[68] P. Junod, J. Rinaldini, J. Wehrli and J. Michielin, "Obfuscator-LLVM -- Software Protection for the Masses," *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection,* 2015.

[69] C. Ping, X. Xiao, M. Bing and X. Li, "Return-Oriented Rootkit without Returns (on the x86)," in *Information and Communications Security. ICICS 2010*, 2010.

[70] F. B. Cohen, "Operating System Protection Through Program Evoluation," *Computers and Security,* vol. 12, no. 6, pp. 565-584, 1993.

[71] D. A. R. P. Agency, "Cyber Fault-tolerant Attack Recovery (CFAR)," [Online]. Available: https://www.darpa.mil/program/cyber-fault-tolerant-attack-recovery. [Accessed 23 10 2022].

[72] T. J. McCabe, "A Complexi-ty Measure," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING,,* vol. 2, no. 4, pp. 308 - 320, December 1976.

GLOSSARY

DEFINITIONS

This section defines the terms used in the context of this document. The intention of Table 5 is to assist the user in their understanding of the document.

Table 5: Definition of Terms Used

TERMS	**Definition**
ROP Gadget	A contiguous instruction sequence already present in the program ending with the RET instruction.
ROP Chain	The linking together of more than one ROP gadget in such a way that a logical goal is achieved.
Zero-Day Vulnerability	Newly discovered software vulnerability that has not been patched.

ACRONYMS

Table 6 lists the acronyms used in this document.

Table 6: Acronyms

TERM	**Definition**
API	Application Programming Interface
ASLR	Address Space Layout Randomization
DEP	Data Execution Prevention
GCC	GNU Compiler Collection
IoT	Internet of Things
O.S.	Operating System
ROP	Return Oriented Programming
V.M.	Virtual Machine
I.R.	Intermediate Representation

APPENDIX A

EXAMPLES OF ROP GADGETS

Table 7 provides examples and descriptions of common ROP gadgets.

Table 7: ROP Gadgets

ROP Gadget	Details
mov qword ptr [r13], r12; ret;	Move the value stored in r12 and store it in the address pointed to by r13. Finally, jump to the following address on the stack
pop r12; pop r13; ret;	Store the value on top of the stack onto the r12 register. Then store the next value in the r13 register. Finally, jump to the following address on the stack.
add eax, ebp; ret;	Add the value ebp to eax, and store the result in eax. Finally, jump to the following address on the stack.
xor byte ptr [r15], r14b; ret;	XOR the lower byte stored in r14, with the value of the byte stored in the address pointed to by r15. Finally, jump to the following address on the stack.
call rax;	Call the address that is stored in the rax register

APPENDIX B

EXAMPLES OF SYSTEM CALLS

Table 8Table 7 lists the system call information needed to execute widespread system calls using Linux x86 systems. The whole system call table can is found at: https://syscall.sh

Table 8: Exit System Call Values

System Call	EAX Value	EAX Value	EBX Value	ECX Value	EDX Value	ESI Value	EDI Value	EBP Value
sys_exit	0x01	Int error_code	-	-	-	-	-	-
sys_read	0c03	unsigned int fd	char __user *buf	size_t count	-	-	-	-
sys_write	0x04	unsigned int fd	const char __user *buf	size_t count	-	-	-	-
sys_execve	0x0b	const char __user *filename	const char __user *const __user *argv	const char __user *const __user *envp	-	-	-	-

APPENDIX C

Figure 24: Shred CPU Clock Cycles Plot

APPENDIX D

Figure 25: Shred ROP Gadgets Plot

APPENDIX E

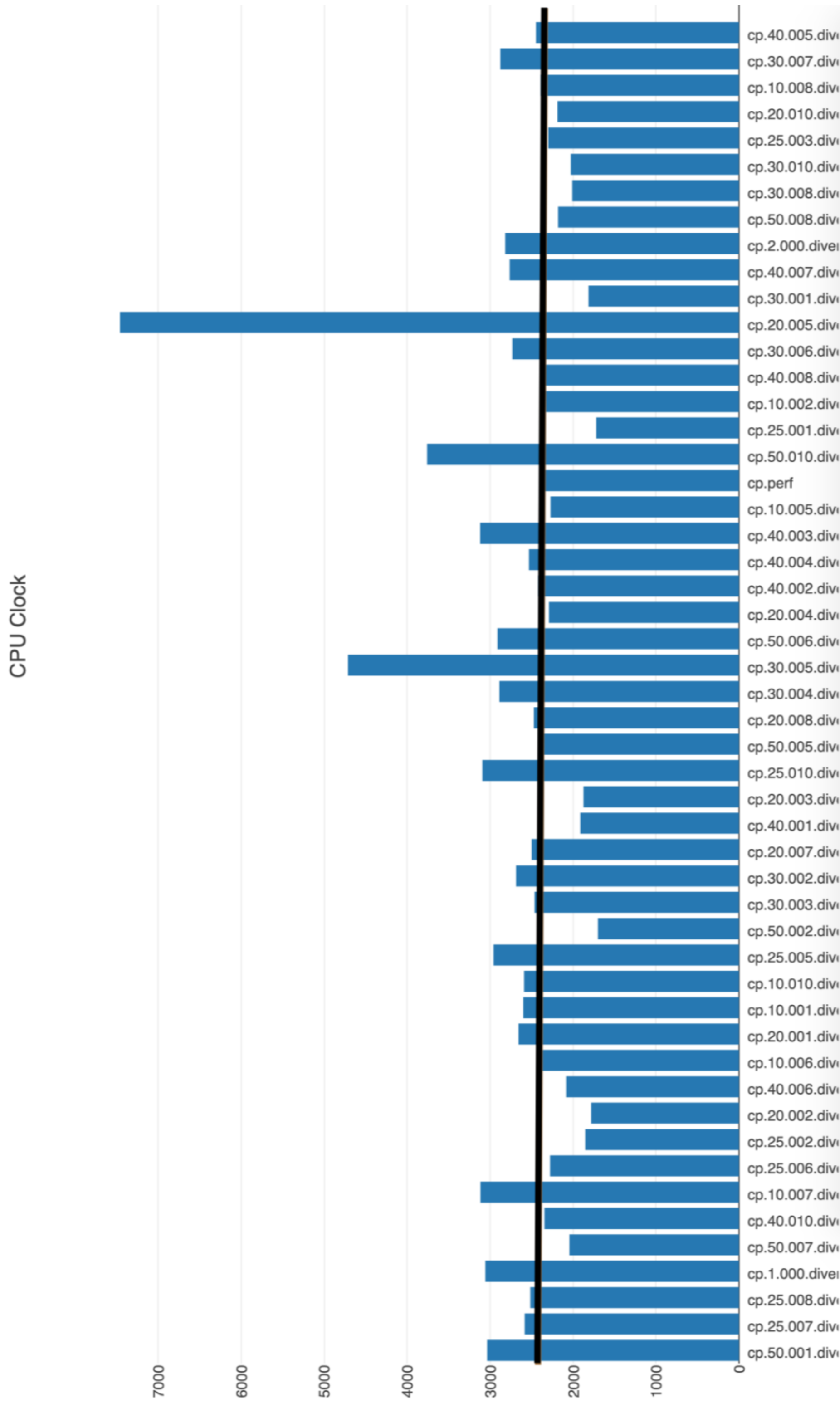

Figure 26: CP CPU Clock Cycles Plot

APPENDIX F

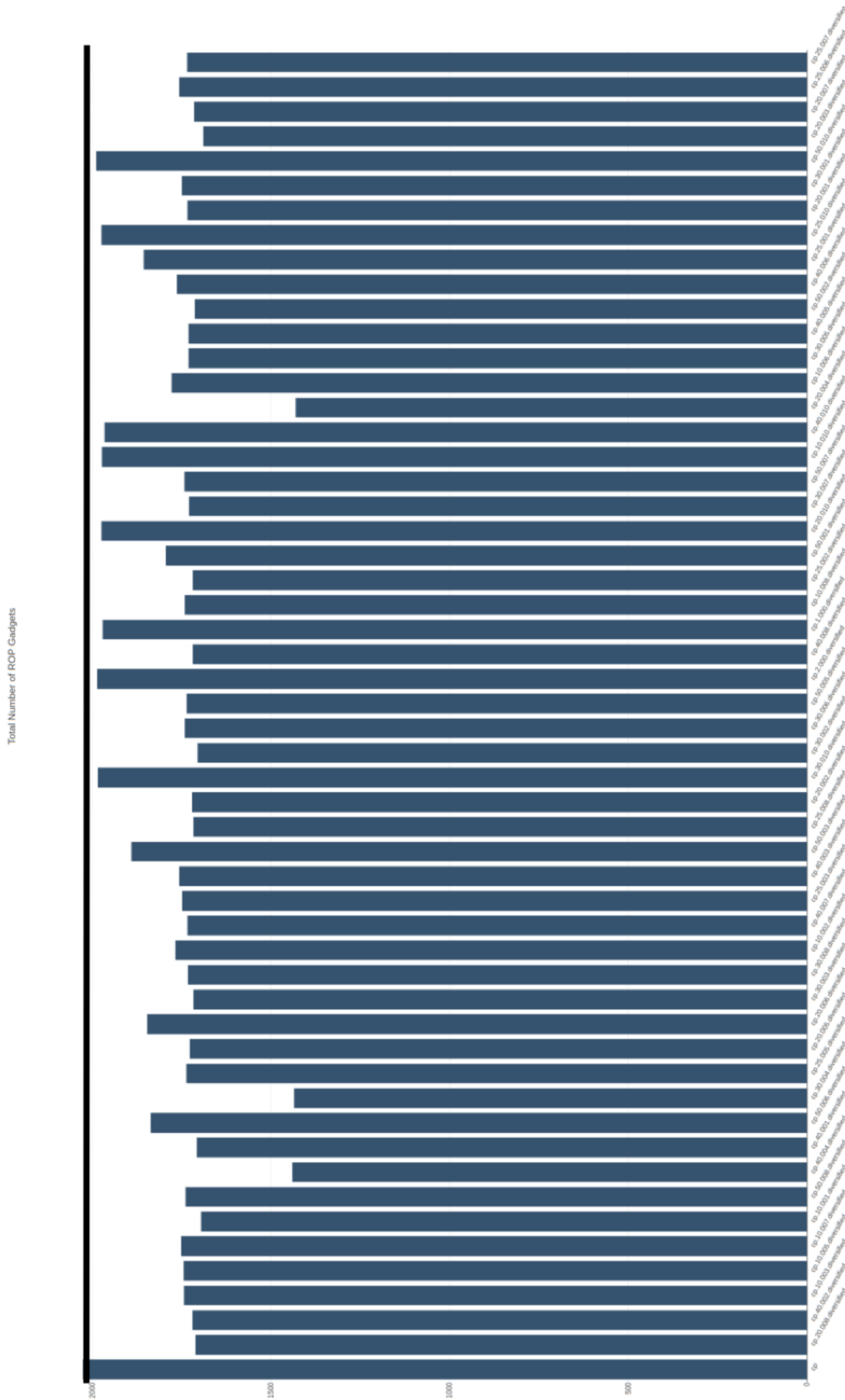

Figure 27: CP ROP Gadgets Plot

APPENDIX G

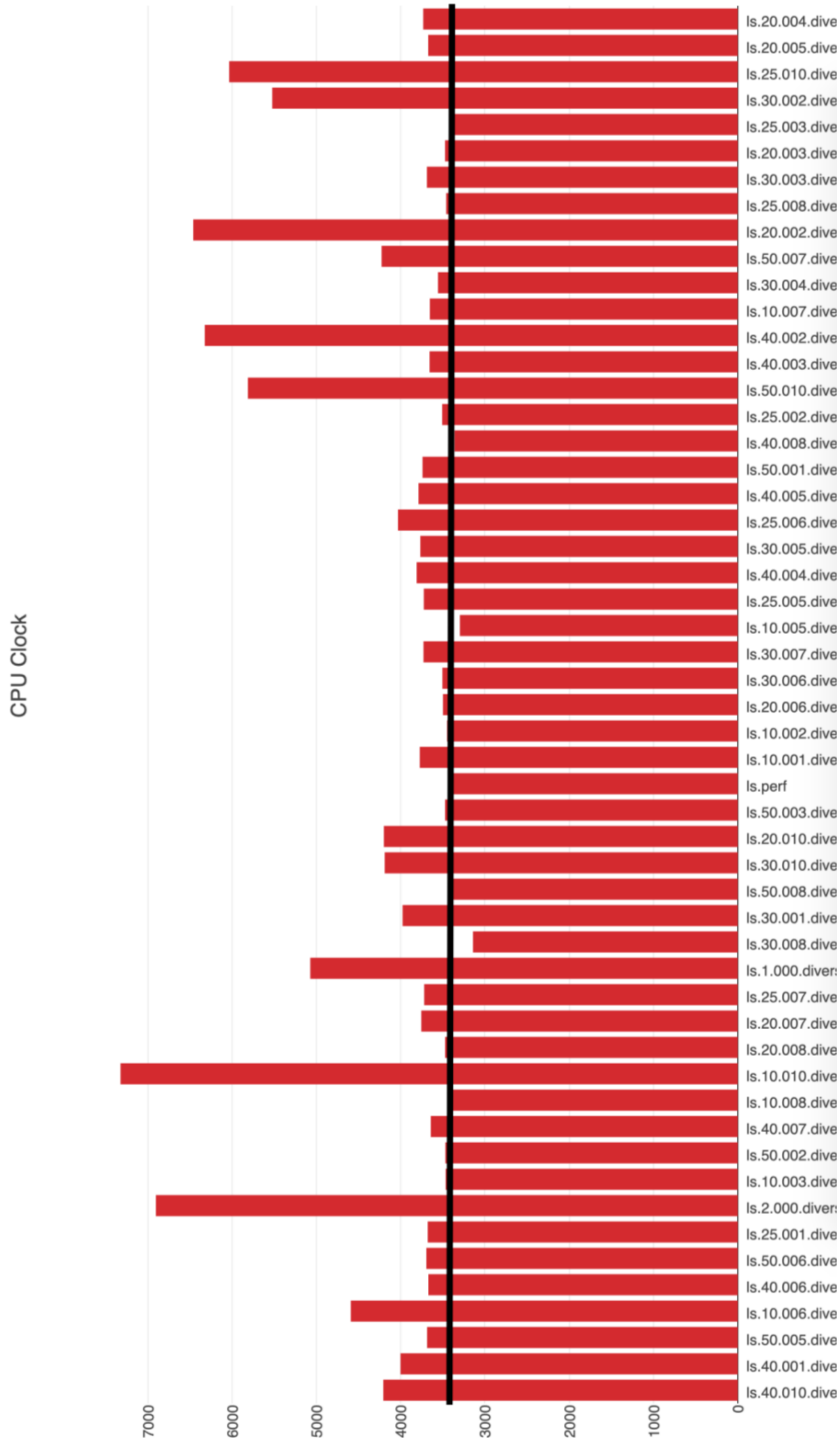

Figure 28: ls CPU Clock Cycles Plot

APPENDIX H

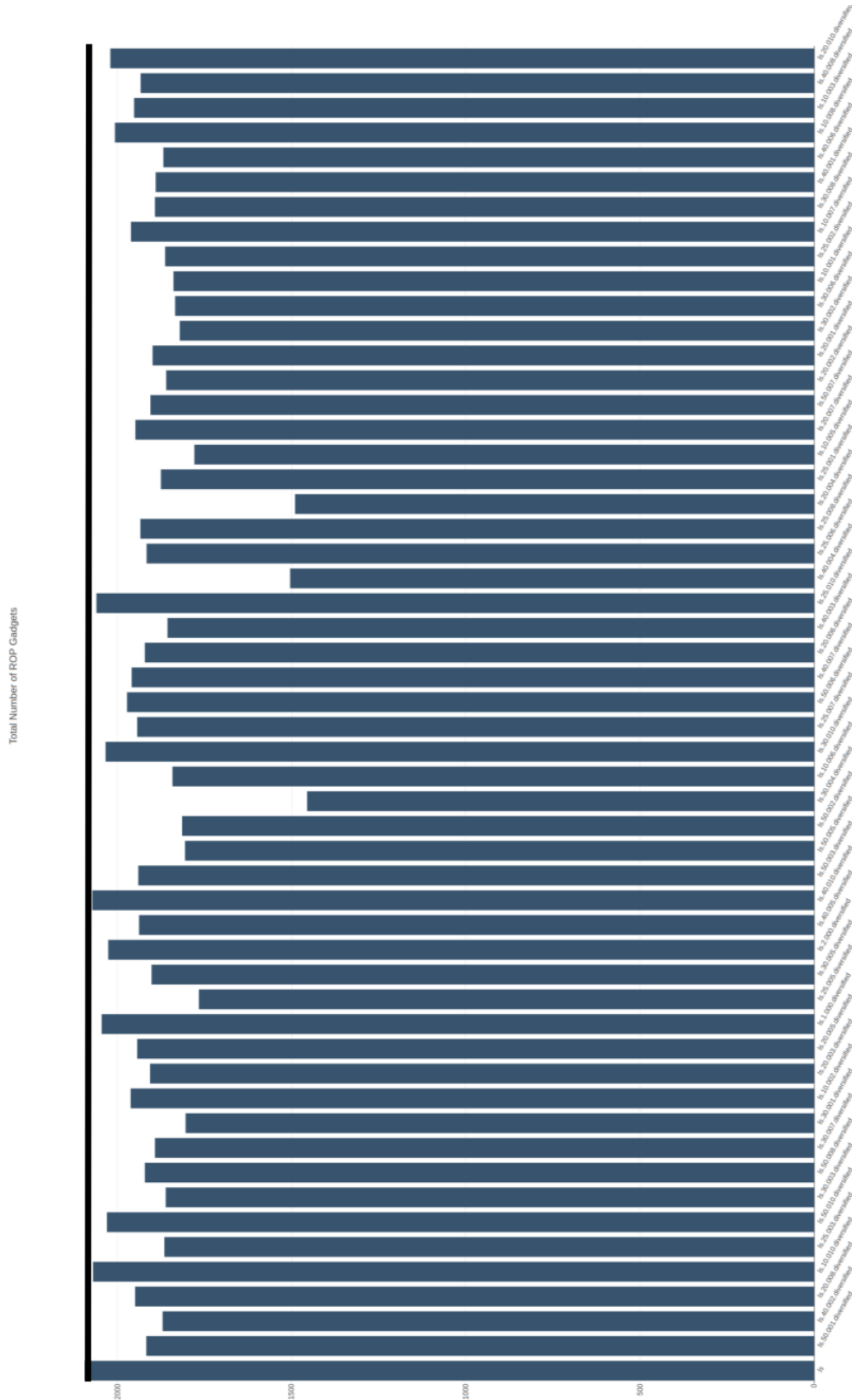

Figure 29: ls ROP Gadgets Plot

APPENDIX I

COMPLETE ATTACK RESISTANCE RESULTS

Table 9 lists complete Attack Resistance scores for all binaries in the dataset used in this work

Table 9: Attack Resistance Score Results for All Binaries
(Refer to Section 5.2.2 for Calculation Details)

Binary Name	Original	SUB	BCF	FLA	SUB & FLA	SUB & BCF	BCF & FLA	ALL
OpenSSL	0	-0.18 - -0.19	-0.31 - -0.33	-0.31 - - 0.34	-0.44 - - 0.47	-2.14 - - 2.19	-1.1 - - 1.16	-2.02 - -2.18
crossfire	0	-0.32 - -0.36	-4.23 - -4.42	-0.24 - - 0.29	-0.48 - - 0.56	-5.9 - - 6.14	-3.61 - - 4.21	-7.27 - -7.56
dnstracker	0	-0.02 - -0.17	-2.95 - -3.85	-0.15 - - 0.32	-0.23 - - 0.5	-4.69 - - 5.73	-3.03 - - 4.03	-5.41 - -7
lamahub	0	-0.37 - -0.45	-5.35 - -5.59	-0.37 - - 0.52	-0.76 - - 0.97	-7.3 - - 8.2	-4.34 - - 4.98	-9.58 - -10.57
mcrypt	0	-0.7 - - 0.91	-8.15 - -8.77	-0.48 - - 0.98	-1.19 - - 1.42	-10.74 - -12.08	-7.18 - - 8.17	-14.39 - - 16.02
mp3info	0	-0.18 - -0.31	-3.75 - -5.12	-0.14 - - 0.45	-0.19 - - 0.75	-6.13 - - 8.12	-7.89 - - 7.89	-7.89 - -7.89
netperf	0	-0.04 - -0.14	-4.44 - -4.85	-0.13 - - 0.21	-0.3 - - 0.35	-6.25 - - 6.85	-3.97 - - 4.49	-8.4 - - 8.96
pdfresurrect	0	-0.25 - -0.38	-6.22 - -7.55	-0.27 - - 0.47	-0.49 - - 0.78	-8.21 - - 9.62	-5.26 - - 11.88	-9.64 - -11.85
Sipp	0	-0.06 - -0.07	-0.12 - -0.18	-0.01 - - 0.02	-0.07 - - 0.08	-0.21 - - 0.4	-0.1 - - 0.15	-0.3 - - 0.39
ytree	0	-0.48 - -0.56	-7.34 - -7.9	-0.41 - - 0.54	-0.89 - - 1.03	-10.1 - - 10.74	-5.89 - - 6.3	-13.2 - -13.75
[	0	-0.17 - -0.27	-5.92 - -7.62	-1.24 - - 1.38	-1.39 - - 1.61	-8.78 - - 9.6	-6.36 - - 7.72	-12.71 - - 13.56
basenc	0	-0.08 - -0.18	-6.17 - -7.12	-1.43 - - 1.66	-1.53 - - 1.73	-8.25 - - 9.28	-5.99 - - 6.69	-12.44 - - 13.28
chown	0	-0.1 - - 0.18	-6.02 - -6.9	-0.81 - - 0.94	-1.03 - - 1.2	-8.55 - - 9.48	-5.75 - - 6.92	-11.72 - - 12.51

csplit	0	-0.66 - -0.76	-8.33 - -9.04	-1.2 - - 1.34	-1.64 - - 1.85	-11.15 - -12.15	-7.36 - - 7.87	-15.02 - - 15.49
dir	0	-0.12 - -0.19	-5.39 - -5.78	-0.7 - - 0.8	-0.89 - - 1.03	-7.35 - - 8.36	-5.1 - - 5.58	-10.31 - - 11.09
env	0	-0.14 - -0.24	-6.58 - -7.84	-1.57 - - 1.77	-1.63 - - 1.96	-9.28 - - 10.07	-6.97 - - 7.99	-13.26 - - 14.74
fmt	0	0.01 - - 0.07	-5.4 - - 6.45	-1.03 - - 1.22	-1.19 - - 1.43	-7.9 - - 8.76	-5.44 - - 6.41	-11.21 - - 12.13
id	0	-0.04 - -0.12	-5.96 - -7.03	-1.37 - - 1.53	-1.39 - - 1.62	-8.33 - - 9.43	-6.54 - - 8	-12.55 - - 13.79
ln	0	-0.08 - -0.13	-5.41 - -6.24	-0.85 - - 0.99	-0.97 - - 1.17	-7.78 - - 8.36	-5.48 - - 5.94	-10.86 - - 11.65
mkfifo	0	-0.24 - -0.31	-6.11 - -7.37	-1.45 - - 1.67	-1.61 - - 1.9	-8.81 - - 9.93	-6.55 - - 7.39	-12.94 - - 13.84
nl	0	-0.75 - -0.83	-8.55 - -9.21	-1.16 - - 1.26	-1.63 - - 1.83	-11.49 - -11.9	-7.61 - - 8.42	-15.3 - -15.88
paste	0	-0.04 - -0.16	-6.6 - - 7.84	-1.57 - - 1.78	-1.62 - - 1.97	-9.09 - - 10.38	-6.82 - - 7.66	-13.31 - - 14.42
printf	0	-0.17 - -0.24	-6.04 - -7.23	-1.37 - - 1.54	-1.47 - - 1.76	-8.77 - - 9.72	-6.54 - - 7.22	-12.83 - - 13.47
rm	0	-0.13 - -0.25	-6.13 - -6.82	-0.81 - - 0.99	-1.03 - - 1.17	-8.64 - - 9.4	-5.72 - - 6.5	-11.75 - - 12.45
sha224sum	0	-1.43 - -1.55	-3.76 - -4.64	-0.58 - - 0.71	-2.3 - - 2.65	-7.48 - - 8.29	-3.99 - - 4.46	-9.31 - -11.05
shuf	0	-0.04 - -0.11	-4.59 - -5.22	-0.87 - - 0.94	-0.83 - - 1.02	-6.65 - - 7.28	-4.48 - - 5.08	-9.01 - -9.96
stdbuf	0	0.00 - - 0.18	-5.35 - -6.52	-1.13 - - 1.3	-1.18 - - 1.46	-7.8 - - 8.59	-5.36 - - 6.55	-11.39 - - 12.23
tail	0	-0.35 - -0.44	-6.35 - -7.08	-1.05 - - 1.33	-1.22 - - 1.4	-8.63 - - 9.32	-6.27 - - 6.75	-11.96 - - 13.28
tr	0	-0.07 - -0.19	-5.88 - -7.09	-1.21 - - 1.37	-1.27 - - 1.46	-8.81 - - 9.54	-5.91 - - 7	-12.31 - - 13.54

uname	0	-0.11 - -0.24	-6.51 - -7.93	-1.85 - - 2.1	-1.74 - - 1.99	-9.35 - - 10.49	-6.83 - - 7.77	-13.64 - - 14.72
users	0	-0.06 - -0.16	-6.37 - -7.04	-1.69 - - 1.91	-1.53 - - 1.79	-8.21 - - 9.33	-6.2 - - 7.31	-12.14 - - 13.75
yes	0	0 - - 0.09	-6.01 - -6.79	-1.35 - - 1.62	-1.55 - - 1.79	-7.85 - - 8.93	-5.98 - - 6.77	-11.82 - - 13.16
b2sum	0	-1.05 - -1.4	-4.42 - -5.11	-0.69 - - 0.78	-2 - - 2.23	-7.74 - - 8.48	-4.27 - - 4.84	-10.25 - - 11.29
cat	0	-0.14 - -0.24	-5.9 - - 7.06	-1.5 - - 1.74	-1.6 - - 1.76	-8.86 - - 9.7	-6.24 - - 7.16	-12.33 - - 13.65
chroot	0	0.15 - 0.01	-5.04 - -6.02	-0.9 - - 1.05	-0.93 - - 1.22	-7.24 - - 8.15	-5.4 - - 6.26	-10.59 - - 11.41
cut	0	-0.16 - -0.26	-6.47 - -7.91	-1.45 - - 1.63	-1.6 - - 1.88	-9.25 - - 10.35	-7.06 - - 8.02	-13.74 - - 15.15
dircolors	0	-0.18 - -0.29	-6.05 - -7.15	-1.39 - - 1.55	-1.54 - - 1.81	-8.64 - - 9.58	-6.15 - - 7.15	-12.31 - - 13.61
expand	0	-0.08 - -0.19	-6.28 - -7.23	-1.46 - - 1.65	-1.51 - - 1.76	-8.83 - - 9.78	-6.57 - - 7.36	-12.85 - - 14.11
fold	0	0.08 - - 0.03	-5.27 - -6.39	-1.19 - - 1.36	-1.21 - - 1.48	-7.77 - - 8.62	-5.44 - - 6.4	-11.12 - - 12.37
install	0	-0.19 - -0.25	-5.64 - -6.41	-0.68 - - 0.79	-0.93 - - 1.06	-8.09 - - 8.82	-5.44 - - 6.07	-11.12 - - 12.04
logname	0	-0.05 - -0.14	-5.83 - -7.05	-1.48 - - 1.74	-1.76 - - 2.03	-8.43 - - 9.34	-6.24 - - 7.16	-12.32 - - 13.51
mknod	0	-0.12 - -0.27	-5.55 - -6.69	-1.25 - - 1.42	-1.34 - - 1.65	-7.99 - - 8.82	-5.66 - - 6.6	-11.74 - - 12.64
nohup	0	-0.06 - -0.21	-5.74 - -6.95	-1.41 - - 1.6	-1.44 - - 1.74	-8.19 - - 9.1	-6.22 - - 7.04	-11.93 - - 13.32
pathchk	0	-0.14 - -0.22	-6.76 - -7.87	-1.87 - - 2.13	-1.76 - - 1.97	-9.47 - - 10.4	-7.09 - - 8.01	-13.73 - - 14.85

ptx	0	-0.68 - -0.79	-7.59 - -8.77	-0.98 - - 1.06	-1.4 - - 1.57	-10.57 - -11.57	-6.98 - - 7.94	-13.97 - -14.4
rmdir	0	-0.07 - -0.17	-6.36 - -7.26	-1.5 - - 1.74	-1.56 - - 1.84	-8.7 - - 9.52	-6.45 - - 7.17	-12.9 - -13.79
sha256sum	0	-1.45 - -1.56	-3.7 - - 4.69	-0.59 - - 0.7	-2.36 - - 2.68	-7.41 - - 8.37	-3.99 - - 4.56	-9.51 - -11.19
sleep	0	-0.07 - -0.17	-5.9 - - 6.68	-1.53 - - 1.74	-1.37 - - 1.65	-7.83 - - 8.91	-5.89 - - 6.69	-11.81 - - 12.73
sty	0	-0.16 - -0.25	-6.04 - -7.11	-1.22 - - 1.35	-1.46 - - 1.73	-9.06 - - 9.66	-5.86 - - 6.98	-12.83 - - 13.71
tee	0	0.04 - - 0.03	-5.62 - -6.66	-1.26 - - 1.43	-1.28 - - 1.52	-7.95 - - 9.17	-5.88 - - 6.75	-11.74 - - 12.69
TRUE	0	-0.2 - - 0.3	-6.26 - -8.15	-1.66 - - 1.9	-1.71 - - 2.01	-9.15 - - 10.07	-6.8 - - 7.79	-13.39 - - 15.23
unexpand	0	-0.06 - -0.14	-6.36 - -7.42	-1.6 - - 1.8	-1.59 - - 1.83	-8.92 - - 9.83	-6.65 - - 7.3	-13.07 - - 14.33
vdir	0	-0.12 - -0.19	-5.39 - -5.78	-0.7 - - 0.8	-0.89 - - 1.03	-7.35 - - 8.36	-5.1 - - 5.58	-10.31 - - 11.09
base32	0	-0.26 - -0.37	-7.54 - -9.14	-1.97 - - 2.23	-2.08 - - 2.3	-10.39 - -11.44	-7.69 - - 8.64	-15.52 - - 16.61
chcon	0	-0.11 - -0.21	-6.02 - -7.03	-0.86 - - 0.99	-1.05 - - 1.23	-8.79 - - 9.63	-5.8 - - 6.6	-11.8 - -12.56
cksum	0	-1.98 - -2.12	-2.31 - -2.73	0.16 - 0.10	-2.39 - - 2.57	-6.23 - - 6.76	-1.73 - - 2.17	-7.39 - -8.01
date	0	-0.72 - -0.96	-13.61 - - 14.58	-3.74 - - 4.16	-4.21 - - 4.5	-16.83 - -17.89	-12.1 - - 13.47	-24.83 - - 26.08
dirname	0	-0.08 - -0.17	-5.99 - -7.41	-1.5 - - 1.73	-1.78 - - 2.12	-8.59 - - 9.34	-6.48 - - 7.6	-12.6 - -13.91
expr	0	-0.65 - -0.73	-8.05 - -8.71	-1.06 - - 1.14	-1.47 - - 1.67	-10.94 - -11.36	-7.14 - - 7.67	-14.49 - - 15.05
groups	0	-0.06 - -0.15	-6.18 - -8.25	-1.47 - - 1.65	-1.5 - - 1.77	-8.85 - - 10.29	-6.84 - - 7.4	-12.83 - - 14.27

join	0	0.18 - 0.05	-4.14 - -5.1	-0.65 - - 0.77	-0.76 - - 0.94	-6.16 - - 7.25	-4.56 - - 5.13	-9.13 - -10.37
ls	0	-0.12 - -0.19	-5.39 - -5.78	-0.7 - - 0.8	-0.89 - - 1.03	-7.35 - - 8.36	-5.1 - - 5.58	-10.31 - - 11.09
mktemp	0	-0.13 - -0.21	-6.42 - -7.62	-1.61 - - 1.84	-1.63 - - 1.94	-9.13 - - 10.23	-6.79 - - 7.77	-13.53 - - 14.48
nproc	0	0.08 - - 0.01	-5.3 - - 6.6	-1.14 - - 1.36	-1.21 - - 1.5	-7.77 - - 9	-5.49 - - 6.48	-11.55 - - 12.43
pinky	0	-0.23 - -0.29	-7.59 - -8.69	-2.03 - - 2.28	-2.17 - - 2.41	-10.03 - -11.22	-8.06 - - 8.82	-15.23 - - 16.37
pwd	0	-0.08 - -0.18	-6.56 - -7.61	-1.55 - - 1.77	-1.64 - - 1.9	-8.87 - - 10.27	-6.67 - - 7.58	-13.46 - - 14.36
runcon	0	-0.15 - -0.26	-6.51 - -7.8	-1.69 - - 1.93	-1.9 - - 2.2	-9.28 - - 10.22	-7.01 - - 8.14	-13.48 - - 15.04
sha384sum	0	-0.18 - -0.4	-2.93 - -3.47	-0.31 - - 0.39	-0.91 - - 1.08	-5.28 - - 6.4	-3.02 - - 3.64	-7.21 - -7.77
sort	0	-0.43 - -0.55	-5.38 - -5.94	-0.57 - - 0.69	-1.12 - - 1.26	-8.11 - - 8.38	-4.97 - - 5.53	-10.42 - - 11.98
sum	0	-0.09 - -0.17	-5.39 - -6.15	-1.13 - - 1.27	-1.21 - - 1.51	-7.69 - - 8.74	-5.52 - - 6.13	-10.87 - - 12.12
test	0	-0.23 - -0.36	-6.53 - -7.82	-1.38 - - 1.71	-1.58 - - 1.87	-9.4 - - 10.39	-6.91 - - 8.55	-13.54 - -16.1
truncate	0	-0.19 - -0.29	-8 - - 9.55	-2.11 - - 2.4	-2.25 - - 2.61	-10.67 - -12.14	-8.29 - - 9.22	-16.03 - - 17.55
uniq	0	0.16 - 0.00	-4.77 - -5.84	-0.97 - - 1.11	-0.99 - - 1.3	-7.11 - - 7.81	-5.12 - - 6.25	-10.2 - -11.31
wc	0	-0.17 - -0.29	-5.68 - -6.48	-1.23 - - 1.41	-1.35 - - 1.59	-8.32 - - 9.49	-6.19 - - 6.96	-11.83 - - 13.11
base64	0	-0.22 - -0.34	-7.34 - -8.78	-1.93 - - 2.18	-2.06 - - 2.44	-10.32 - -11.38	-7.64 - - 8.46	-15.22 - - 16.37
chgrp	0	-0.12 - -0.21	-6.05 - -6.86	-0.88 - - 1	-1.05 - - 1.21	-8.73 - - 9.42	-5.92 - - 6.65	-11.75 - - 12.54

comm	0	-0.03 - -0.65	-5.6 - - 6.99	-1.24 - - 1.45	-1.29 - - 1.58	-8.12 - - 9.02	-6.07 - - 6.86	-11.71 - - 13.08
dd	0	-0.07 - -0.16	-4.72 - -5.37	-0.67 - - 0.81	-0.83 - - 1.06	-7.22 - - 7.85	-4.99 - - 5.61	-9.93 - -10.54
du	0	-0.57 - -0.61	-7.72 - -10.03	-0.94 - - 1.03	-1.34 - - 1.48	-10.45 - -10.92	-6.78 - - 7.2	-13.86 - - 14.48
factor	0	-0.19 - -0.3	-4.48 - -5.33	-0.65 - - 0.75	-0.81 - - 0.97	-6.79 - - 7.92	-4.58 - - 4.91	-9.1 - - 9.69
head	0	-0.12 - -0.2	-6.12 - -7.34	-1.3 - - 1.49	-1.38 - - 1.6	-8.79 - - 9.48	-6.04 - - 7.48	-12.51 - - 13.39
kill	0	-0.06 - -0.19	-6.42 - -7.64	-1.48 - - 1.67	-1.62 - - 1.92	-8.84 - - 10.03	-6.54 - - 7.55	-13.04 - - 14.26
md5sum	0	-0.81 - -0.98	-5.86 - -6.88	-1.21 - - 1.41	-2.13 - - 2.45	-9.11 - - 10.31	-6.04 - - 6.52	-12.54 - -13.5
mv	0	-0.27 - -0.34	-6.19 - -6.56	-0.72 - - 0.83	-0.95 - - 1.05	-8.48 - - 9.3	-5.73 - - 6.41	-11.62 - - 12.57
numfmt	0	-0.06 - -0.13	-5.85 - -6.66	-0.99 - - 1.15	-1.15 - - 1.41	-8.12 - - 8.8	-6.1 - - 6.62	-11.74 - - 12.37
pr	0	-0.18 - -0.39	-8.21 - -9.35	-1.71 - - 1.89	-2.01 - - 2.23	-11.5 - - 12.31	-7.72 - - 8.97	-16.85 - - 18.13
readlink	0	-0.09 - -0.19	-5.95 - -6.88	-1.09 - - 1.23	-1.23 - - 1.39	-8.14 - - 9.03	-5.95 - - 6.66	-11.51 - - 12.61
seq	0	-0.11 - -0.22	-6.09 - -7.3	-1.38 - - 1.62	-1.51 - - 1.87	-9.01 - - 9.85	-6.81 - - 7.43	-13.06 - - 14.03
sha512sum	0	-0.17 - -0.41	-2.96 - -3.56	-0.3 - - 0.39	-0.94 - - 1.09	-5.42 - - 6.32	-3.14 - - 3.73	-7.36 - -8.11
split	0	-0.02 - -0.09	-5.55 - -6.25	-1.07 - - 1.31	-1.09 - - 1.26	-7.94 - - 8.49	-5.52 - - 6.49	-11.02 - - 12.03
sync	0	-0.08 - -0.16	-6.77 - -7.91	-1.89 - - 2.1	-1.69 - - 2	-9.11 - - 10.14	-6.95 - - 7.9	-13.31 - -14.6
timeout	0	-0.12 - -0.22	-5.7 - - 6.78	-1.31 - - 1.48	-1.38 - - 1.61	-8.24 - - 9.07	-6.09 - - 6.82	-11.91 - - 13.18

tsort	0	-0.07 - -0.13	-5.88 - -6.91	-1.3 - -1.47	-1.33 - -1.53	-8.4 - -9.23	-6.07 - -6.82	-11.62 - -12.88
unlink	0	-0.04 - -0.14	-5.82 - -7.82	-1.48 - -1.71	-1.65 - -1.94	-8.28 - -9.28	-6.2 - -7.89	-12.2 - -13.55
who	0	-0.17 - -0.31	-6.71 - -7.89	-1.69 - -1.89	-1.71 - -1.94	-9.29 - -10.3	-6.82 - -7.78	-13.01 - -14.45
basename	0	-0.03 - -0.12	-5.8 - -7	-1.58 - -1.83	-1.55 - -1.77	-8.43 - -9.14	-6.15 - -7.42	-12.25 - -13.45
chmod	0	-0.24 - -0.72	-6.27 - -6.99	-0.86 - -1	-1.15 - -1.29	-8.82 - -9.59	-5.81 - -6.55	-11.92 - -12.91
cp	0	-0.16 - -0.23	-5.72 - -6.21	-0.75 - -0.87	-0.92 - -1.06	-8.16 - -9.02	-5.47 - -6.09	-11.1 - -11.92
df	0	-0.19 - -0.29	-5.94 - -6.57	-0.83 - -1	-1.07 - -1.24	-8.17 - -9.02	-5.7 - -6.18	-11.49 - -11.89
echo	0	-0.1 - -0.18	-6.38 - -7.8	-1.83 - -2.06	-1.69 - -1.94	-9.1 - -10.06	-7.11 - -7.87	-13.41 - -14.72
FALSE	0	-0.19 - -0.29	-6.35 - -7.91	-1.64 - -1.87	-1.68 - -1.98	-9.09 - -10.02	-6.82 - -7.76	-13.29 - -15.13
hostid	0	-0.05 - -0.14	-5.82 - -7.25	-1.46 - -1.7	-1.64 - -1.92	-8.28 - -9.17	-6.24 - -7.05	-12.17 - -13.37
link	0	-0.03 - -0.14	-5.85 - -7	-1.45 - -1.68	-1.63 - -1.91	-8.26 - -9.14	-6.19 - -7.23	-12.04 - -13.3
mkdir	0	-0.13 - -0.2	-5.57 - -6.71	-1.06 - -1.27	-1.19 - -1.46	-8.05 - -8.86	-5.97 - -6.71	-11.82 - -12.67
nice	0	-0.09 - -0.2	-6.7 - -7.91	-1.56 - -1.82	-1.69 - -2.01	-9.22 - -10.17	-6.83 - -8.06	-13.63 - -14.92
od	0	0.15 - 0.09	-4.56 - -5.32	-0.96 - -1.11	-0.81 - -1.02	-6.48 - -7.2	-4.84 - -5.24	-9.12 - -9.92
printenv	0	-0.05 - -0.19	-6.37 - -7.77	-1.58 - -1.83	-1.77 - -2.05	-8.95 - -9.83	-6.68 - -7.83	-13.2 - -14.41
realpath	0	-0.16 - -0.26	-6.21 - -6.83	-1.14 - -1.27	-1.3 - -1.51	-8.32 - -9.67	-6.07 - -6.66	-11.79 - -13.02

sha1sum	0	-1.54 - -1.85	-6.28 - -8.1	-1.37 - - 1.56	-2.98 - - 3.22	-9.67 - - 11.34	-6.22 - - 6.98	-13.58 - - 15.54
shred	0	-0.22 - -0.3	-5.16 - -6.05	-0.82 - - 0.99	-1.1 - - 1.32	-7.39 - - 8.1	-5.07 - - 5.73	-10.38 - - 11.18
stat	0	-0.13 - -0.26	-6.28 - -7.13	-1.19 - - 1.33	-1.34 - - 1.57	-8.74 - - 9.42	-6.17 - - 7.06	-12.25 - - 13.08
tac	0	-0.66 - -0.76	-8.14 - -8.82	-1.07 - - 1.17	-1.54 - - 1.72	-10.99 - -11.46	-7.23 - - 7.75	-14.54 - - 15.11
touch	0	-0.64 - -0.77	-12.29 - - 13.29	-3.52 - - 3.88	-3.91 - - 4.1	-15.28 - -16.31	-11.06 - -12.19	-22.44 - - 23.48
tty	0	-0.07 - -0.17	-6.4 - - 7.75	-1.66 - - 1.9	-1.69 - - 2	-9.18 - - 10.29	-6.9 - - 7.86	-13.55 - - 14.77
uptime	0	-0.05 - -0.33	-7.34 - -8.45	-1.55 - - 1.73	-1.63 - - 1.94	-9.88 - - 11.23	-7.03 - - 7.98	-14.71 - - 16.33
whoami	0	-0.03 - -0.11	-5.85 - -7.04	-1.45 - - 1.66	-1.7 - - 1.97	-8.15 - - 9.3	-6.11 - - 6.96	-12.06 - - 13.23

APPENDIX J

COMPLETE EXPLOIT COMPLEXITY SCORE RESULTS

Table 10 lists complete Exploit Complexity scores for all binaries in the dataset used in this work

Table 10: Exploit Complexity Score Results for All Binaries
(Refer to Section 5.2.3 for Calculation Details)

Binary Name	Original	SUB	BCF	FLA	SUB & FLA	SUB & BCF	BCF & FLA	ALL
OpenSSL	15393	15944 - 16046	18358 - 18803	22258 - 22339	22562 - 22820	31809 - 32675	29895 - 30915	31028 - 31518
crossfire	3503	3831 - 3943	7653 - 8108	3590 - 3667	4001 - 4124	14841 - 15818	9247 - 9502	20450 - 21569
dnstracker	62	50 – 70	111 – 154	50 – 66	52 - 75	276 - 305	127 – 189	359 – 443
lamahub	55	89 – 114	651 – 765	53 – 61	102 – 143	1599 – 1813	759 – 890	2313 – 2438
mcrypt	28	51 – 77	416 – 543	30 – 49	62 – 87	1081 – 1239	662 – 799	1948 – 2158
mp3info	15	19 – 30	48 – 130	13 – 19	17 – 36	249 – 340	340 – 340	340 - 340
netperf	165	241 - 285	720 - 906	224 - 261	260 - 304	1902 - 2172	1224 - 1422	3380 - 3758
pdfresurrect	15	14 - 21	119 - 197	15 - 24	14 - 30	333 - 422	202 - 271	533 - 646
Sipp	1586	1657 - 1703	1664 - 1797	1686 - 1697	1666 - 1722	1905 - 2099	1784 - 1972	2148 - 2302
ytree	49	77 – 98	1007 – 1253	66 - 111	98 - 159	2855 - 3028	1284 - 1468	4514 - 4780
[	275	39 – 76	255 - 458	27 – 48	41 – 97	905 - 1024	548 - 683	1640 - 1842
basenc	372	37 – 85	406 - 525	31 – 51	69 - 120	1155 - 1404	749 - 911	2139 - 2377
chown	471	63 – 98	526 - 713	50 – 81	74 - 130	1543 - 1827	864 - 1109	2562 - 2814
csplit	905	224 – 313	1269 - 1494	119 - 154	334 - 414	3065 - 3931	1865 - 2079	5259 - 5472

dir	1019	133 – 214	987 - 1255	104 - 144	196 - 277	2919 - 3425	1595 - 1789	4858 - 5343
env	260	34 – 61	270 - 385	37 – 61	45 – 85	829 – 926	530 – 675	1539 - 1728
fmt	260	33 – 69	237 - 422	28 – 54	55 - 116	842 - 1029	498 - 598	1518 - 1694
id	259	28 – 56	250 - 392	29 – 57	42 – 90	829 - 1006	507 - 605	1551 - 1807
ln	514	54 – 103	571 - 758	40 – 63	79 - 123	1532 - 1827	875 - 983	2575 – 2805
mkfifo	239	27 – 54	200 - 368	28 – 47	36 – 81	719 – 853	437 - 529	1355 - 1504
nl	849	219 - 304	1103 - 1364	118 - 145	322 - 391	2734 - 3056	1724 - 2449	4640 - 4812
paste	229	24 – 53	198 - 358	27 – 46	33 – 76	675 – 835	423 - 540	1269 - 1459
printf	268	35 – 67	251 - 440	28 – 47	42 – 79	810 - 1001	501 - 689	1532 - 1662
rm	536	51 – 99	573 - 746	46 – 74	73 - 119	1657 - 1899	874 - 1002	2732 - 2915
sha224sum	291	102 - 181	300 - 441	31 – 49	111 - 168	934 - 1118	551 - 675	1641 - 1998
shuf	461	65 - 115	470 - 642	56 - 74	90 - 154	1311 - 1550	768 - 876	2299 - 2579
stdbuf	239	26 – 54	218 - 377	29 – 48	37 – 97	788 – 916	471 - 608	1484 - 1638
tail	484	66 - 120	518 - 734	40 – 74	72 - 157	1594 - 1849	893 - 1089	2691 – 2993
tr	289	28 – 60	294 - 478	53 – 74	42 - 100	957 - 1157	641 - 714	1778 - 2005
uname	197	24 – 53	195 - 339	30 – 49	34 – 79	621 – 778	413 - 515	1189 - 1415
users	243	40 - 69	236 - 373	53 - 75	56 - 100	656 - 826	433 - 530	1242 - 1449
yes	241	37 - 66	229 - 345	42 - 62	49 - 89	619 - 794	430 - 536	1231 - 1433

b2sum	319	45 - 82	349 - 518	41 - 60	57 - 116	935 - 1198	609 - 729	1815 - 2051
cat	245	47 - 76	228 - 389	51 - 82	55 - 101	723 - 851	474 - 589	1331 - 1554
chroot	279	25 - 56	242 - 399	29 - 48	36 - 95	846 - 999	519 - 602	1457 - 1676
cut	251	40 - 69	259 - 399	32 - 51	57 - 96	817 - 995	545 - 736	1513 - 1839
dircolors	261	49 - 89	212 - 410	30 - 54	71 - 114	808 - 937	461 - 553	1405 - 1623
expand	236	26 - 55	225 - 378	30 - 55	37 - 78	749 - 897	435 - 550	1362 - 1582
fold	236	26 - 55	207 - 367	28 - 47	34 - 95	663 - 847	457 - 537	1327 - 1492
install	884	128 - 190	945 - 1281	106 - 130	205 - 278	2558 - 2912	1459 - 1674	4393 - 5373
logname	214	39 - 68	206 - 336	40 - 62	46 - 91	621 - 748	422 - 514	1180 - 1344
mknod	255	29 - 60	214 - 385	30 - 51	39 - 101	729 - 882	471 - 571	1406 - 1569
nohup	246	37 - 66	248 - 415	40 - 60	46 - 92	677 - 843	456 - 605	1300 - 1581
pathchk	199	24 - 53	195 - 342	30 - 49	33 - 82	660 - 809	434 - 514	1288 - 1461
ptx	1066	239 - 330	1276 - 1625	132 - 162	339 - 422	3289 - 3753	2002 - 2255	5437 - 5674
rmdir	214	24 - 53	203 - 348	30 - 51	36 - 76	703 - 826	425 - 508	1327 - 1460
sha256sum	290	104 - 181	281 - 421	30 - 57	107 - 170	950 - 1135	563 - 675	1642 - 1849
sleep	256	49 - 91	242 - 389	48 - 68	59 - 113	691 - 849	462 - 549	1382 - 1495
stty	283	29 - 60	330 - 495	33 - 52	43 - 106	931 - 1098	599 - 750	1791 - 1966
tee	273	24 - 53	254 - 407	27 - 48	35 - 76	721 - 922	437 - 550	1360 - 1507

TRUE	186	26 - 55	184 - 323	29 - 48	35 - 78	618 - 747	391 - 490	1134 - 1444
unexpand	233	27 - 56	212 - 361	30 - 58	45 - 92	729 - 912	448 - 546	1348 - 1593
vdir	1019	133 - 214	987 - 1255	104 - 144	196 - 277	2919 - 3425	1595 - 1789	4858 - 5343
base32	271	34 - 65	257 - 458	31 - 51	51 - 99	834 - 957	549 - 645	1591 - 1798
chcon	483	47 - 84	543 - 751	43 - 64	60 - 106	1658 - 1881	840 - 1006	2655 - 2888
cksum	793	343 - 445	736 - 936	82 - 125	363 - 497	2032 - 2240	1019 - 1517	3154 - 3450
date	409	101 - 128	754 - 905	86 - 136	163 - 217	1648 - 1979	1505 - 1707	3872 - 4122
dirname	206	26 - 55	194 - 348	29 - 49	37 - 79	642 - 786	416 - 502	1201 - 1388
expr	927	243 - 323	1246 - 1421	139 - 165	340 - 413	2981 - 3260	1852 - 2049	4943 - 5193
groups	225	24 - 53	209 - 386	30 - 49	33 - 76	686 - 821	431 - 600	1274 - 1550
join	362	37 - 69	297 - 510	33 - 54	45 - 107	973 - 1189	611 - 731	1781 - 1986
ls	1019	133 - 214	987 - 1255	104 - 144	196 - 277	2919 - 3425	1595 - 1789	4858 - 5343
mktemp	233	38 - 64	227 - 395	33 - 50	45 - 85	679 - 849	471 - 556	1319 - 1520
nproc	235	26 - 55	209 - 384	28 - 47	37 - 95	674 - 854	447 - 533	1307 - 1470
pinky	237	29 - 61	263 - 417	35 - 60	38 - 87	796 - 913	546 - 658	1512 - 1756
pwd	205	24 - 53	220 - 355	30 - 49	36 - 79	647 - 833	439 - 565	1319 - 1484
runcon	192	24 - 53	209 - 329	30 - 49	33 - 76	667 - 785	410 - 521	1232 - 1460
sha384sum	305	87 - 121	307 - 470	34 - 54	113 - 137	983 - 1127	572 - 675	1658 - 1805

sort	839	114 - 202	950 - 1113	52 - 87	178 - 256	2575 - 2874	1356 - 1516	4160 - 4441
sum	331	60 - 91	347 - 507	64 - 84	78 - 137	935 - 1208	592 - 711	1740 - 2072
test	263	27 - 69	233 - 438	25 - 40	36 - 89	837 - 962	538 - 689	1618 - 1809
truncate	226	27 - 66	229 - 409	28 - 48	54 - 93	706 - 850	526 - 657	1415 - 1614
uniq	314	27 - 60	248 - 413	36 - 59	45 - 106	860 - 1099	489 - 677	1515 - 1748
wc	318	47 - 101	303 - 478	46 - 66	82 - 139	959 - 1208	554 - 673	1703 - 1962
base64	263	27 - 67	266 - 440	34 - 51	53 - 100	829 - 953	547 - 652	1565 - 1780
chgrp	447	58 - 94	527 - 710	50 - 73	77 - 128	1495 - 1822	816 - 961	2551 - 2771
comm	287	27 - 55	272 - 433	30 - 51	37 - 83	764 - 974	552 - 618	1475 - 1715
dd	469	77 - 112	457 - 627	65 - 90	92 - 156	1299 - 1482	753 - 884	2295 - 2502
du	1346	310 - 403	1696 - 4603	164 - 185	423 - 513	4335 - 4832	2499 - 2789	7232 - 7583
factor	603	69 - 125	488 - 679	70 - 88	106 - 155	1372 - 1560	847 - 991	2288 - 2505
head	255	27 - 55	218 - 462	39 - 61	40 - 101	859 - 1045	552 - 630	1576 - 1756
kill	232	28 - 57	245 - 379	31 - 50	37 - 86	687 - 816	443 - 528	1309 - 1472
md5sum	270	44 - 82	285 - 444	34 - 51	55 - 101	839 - 1044	532 - 679	1591 - 1745
mv	897	123 - 178	1070 - 1316	108 - 134	177 - 236	2806 - 3107	1568 - 1871	4552 - 5070
numfmt	342	33 - 64	348 - 551	31 - 53	48 - 107	1056 - 1243	674 - 802	1940 - 2131
pr	469	42 - 68	473 - 604	62 - 108	79 - 144	1427 - 1667	859 - 983	3010 - 3242

readlink	342	36 - 80	360 - 485	33 - 52	56 - 99	1035 - 1212	584 - 809	1848 - 2012
seq	255	29 - 65	248 - 396	27 - 48	42 - 85	772 - 927	482 - 616	1431 - 1663
sha512sum	296	80 - 119	305 - 485	36 - 55	104 - 135	968 - 1117	520 - 698	1632 - 1827
split	362	37 - 71	352 - 512	39 - 61	57 - 109	993 - 1208	657 - 802	1893 - 2284
sync	204	26 - 53	217 - 353	27 - 46	35 - 78	645 - 773	420 - 503	1235 - 1415
timeout	261	32 - 75	252 - 370	31 - 51	40 - 94	746 - 923	474 - 581	1439 - 1723
tsort	275	47 - 76	253 - 408	51 - 75	61 - 107	716 - 928	484 - 595	1395 - 1576
unlink	215	40 - 69	205 - 344	40 - 60	48 - 90	620 - 779	420 - 509	1167 - 1351
who	235	45 - 74	268 - 440	54 - 75	63 - 104	764 - 923	493 - 592	1387 - 1623
basename	209	26 - 55	220 - 351	32 - 51	37 - 82	656 - 809	413 - 516	1264 - 1490
chmod	454	53 - 88	498 - 730	50 - 70	73 - 127	1489 - 1739	829 - 946	2517 - 2735
cp	779	116 - 173	916 - 1119	109 - 129	172 - 234	2261 - 2694	1345 - 1514	3805 - 4237
df	542	95 - 146	691 - 865	43 - 69	123 - 179	1754 - 2088	996 - 1176	3022 - 3203
echo	199	24 - 53	192 - 347	30 - 49	35 - 77	631 - 753	403 - 510	1206 - 1422
FALSE	190	24 - 53	179 - 325	27 - 46	33 - 76	621 - 733	392 - 501	1134 - 1434
hostid	218	37 - 66	209 - 339	40 - 61	46 - 91	611 - 782	420 - 504	1169 - 1345
link	223	37 - 69	225 - 334	43 - 63	51 - 92	623 - 777	420 - 507	1167 - 1425
mkdir	337	35 - 65	277 - 485	35 - 54	46 - 100	927 - 1084	535 - 635	1703 - 1863

nice	217	25 - 54	210 - 358	27 - 46	34 - 84	657 - 774	425 - 552	1304 - 1489
od	504	48 - 81	369 - 545	35 - 55	68 - 126	1122 - 1299	686 - 808	1926 - 2125
printenv	197	24 - 53	193 - 321	27 - 47	33 - 76	627 - 760	397 - 503	1217 - 1344
realpath	344	39 - 86	376 - 529	36 - 55	67 - 114	1098 - 1277	591 - 745	1878 - 2206
sha1sum	268	46 - 102	302 - 442	35 - 55	72 - 119	872 - 1063	564 - 661	1645 - 1784
shred	413	66 - 99	397 - 608	35 - 52	82 - 144	1116 - 1382	695 - 852	2026 - 2369
stat	467	59 - 94	479 - 675	45 - 90	84 - 121	1493 - 1695	882 - 1067	2647 - 2797
tac	854	217 - 303	1118 - 1444	115 - 154	316 - 388	2806 - 3066	1729 - 2182	4684 - 4868
touch	411	111 - 136	687 - 856	91 - 142	144 - 206	1529 - 1848	1375 - 1834	3449 - 3758
tty	192	24 - 53	189 - 335	27 - 46	33 - 76	617 - 746	393 - 491	1199 - 1325
uptime	312	45 - 75	293 - 456	63 - 85	85 - 117	884 - 1088	578 - 733	1885 - 2046
whoami	215	39 - 68	222 - 350	40 - 60	48 - 89	610 - 766	427 - 512	1182 - 1342

APPENDIX K

COMPLETE RESISTANCE TO REVERSE ENGINEERING SCORE RESULTS

Table 11 lists complete Resistance to Reverse Engineering scores for all binaries in the dataset used in this work

Table 11: Resistance to Reverse Engineering Score Results for All Binaries
(Refer to Section 5.2.4 for Calculation Details)

Binary Name	Original	SUB	BCF	FLA	SUB & FLA	SUB & BCF	BCF & FLA	ALL
OpenSSL	30520	30549 - 30597	63545 - 64765	77792 - 77803	77738 - 77816	65164 - 66292	144340 - 145741	152999 - 155075
crossfire	20911	20905 - 20911	33807 - 34249	33368 - 33374	33368 - 33374	33689 - 34141	52948 - 53372	52488 - 53347
dnstracker	271	271 - 271	489 - 535	545 - 545	545 - 545	507 - 549	873 - 991	878 - 983
lamahub	976	976 - 976	2114 - 2184	2177 - 2177	2177 - 2177	2080 - 2212	3892 - 4031	3907 - 4143
mcrypt	1207	1207 - 1207	2593 - 2687	2625 - 2625	2625 - 2625	2561 - 2663	4715 - 4935	4596 - 4875
mp3info	319	319 - 319	565 - 619	659 - 659	659 - 659	589 - 655	635 - 635	635 - 635
netperf	2222	2222 - 2222	4615 - 4805	4968 - 4968	4968 - 4968	4668 - 4844	8778 - 9033	8792 - 8973
pdfresurrect	345	345 - 345	711 - 773	721 - 721	721 - 721	713 - 765	1281 - 1417	1248 - 1350
Sipp	5909	5909 - 5909	6273 - 6349	6341 - 6341	6341 - 6341	6269 - 6371	6864 - 6981	6859 - 6991
ytree	2578	2578 - 2578	5704 - 5944	5960 - 5960	5960 - 5960	5608 - 5852	10566 - 10949	10638 - 10942
[	663	1356 - 1498	2426 - 2642	2211 - 2211	2211 - 2211	2500 - 2568	4015 - 4126	3971 - 4217
basenc	890	2005 - 2095	3551 - 3731	3009 - 3009	3009 - 3009	3599 - 3689	5473 - 5700	5491 - 5701

chown	1262	1910 - 2088	3646 - 3787	3410 - 3410	3410 - 3410	3659 - 3804	6007 - 6376	6083 - 6314
csplit	3293	3960 - 4055	8037 - 8262	7527 - 7527	7527 - 7527	8076 - 8227	13803 - 14166	13750 - 14104
dir	2909	3389 - 4087	6669 - 6849	6272 - 6272	6272 - 6272	7369 - 8058	11402 - 11658	11342 - 11742
env	670	1286 - 1561	2302 - 2470	2107 - 2107	2107 - 2107	2366 - 2424	3763 - 3891	3764 - 3928
fmt	688	1285 - 1461	2325 - 2469	2113 - 2113	2113 - 2113	2365 - 2441	3757 - 3916	3724 - 3891
id	628	1354 - 1558	2436 - 2546	2192 - 2192	2192 - 2192	2450 - 2554	3883 - 4048	3882 - 4100
ln	1149	1756 - 1967	3306 - 3485	3043 - 3043	3043 - 3043	3317 - 3466	5490 - 5694	5501 - 5719
mkfifo	504	1188 - 1336	2092 - 2242	1876 - 1876	1876 - 1876	2140 - 2188	3346 - 3449	3321 - 3510
nl	3101	3504 - 3601	7199 - 7458	6804 - 6804	6804 - 6804	7290 - 7394	12539 - 12830	12430 - 12693
paste	529	1162 - 1300	2048 - 2190	1827 - 1827	1827 - 1827	2090 - 2166	3277 - 3395	3240 - 3409
printf	625	1283 - 1469	2283 - 2431	2097 - 2097	2097 - 2097	2329 - 2419	3729 - 3867	3714 - 3880
rm	1269	1894 - 2185	3622 - 3777	3347 - 3347	3347 - 3347	3643 - 3773	5978 - 6261	6035 - 6250
sha224sum	682	1312 - 1561	2424 - 2590	2266 - 2266	2266 - 2266	2480 - 2548	4063 - 4161	4089 - 4251
shuf	968	1597 - 1763	2977 - 3146	2756 - 2756	2756 - 2756	3000 - 3133	4921 - 5070	4915 - 5138
stdbuf	562	1223 - 1375	2157 - 2297	1978 - 1978	1978 - 1978	2227 - 2283	3456 - 3636	3509 - 3634
tail	1386	2024 - 2341	3808 - 4018	3723 - 3723	3723 - 3723	3806 - 3931	6495 - 6685	6484 - 6751
tr	759	1413 - 1825	2571 - 2749	2462 - 2462	2462 - 2462	2647 - 2745	4347 - 4519	4343 - 4566
uname	485	1108 - 1245	1940 - 2080	1716 - 1716	1716 - 1716	1982 - 2060	3055 - 3180	3042 - 3198

users	472	1121 - 1228	1977 - 2091	1721 - 1721	1721 - 1721	1999 - 2069	3062 - 3189	3056 - 3235
yes	462	1121 - 1228	1977 - 2091	1721 - 1721	1721 - 1721	1999 - 2069	3062 - 3189	3056 - 3235
b2sum	739	1412 - 1646	2692 - 2882	2577 - 2577	2577 - 2577	2720 - 2816	4617 - 4792	4626 - 4766
cat	541	1209 - 1253	2135 - 2285	1906 - 1906	1906 - 1906	2195 - 2279	3428 - 3510	3416 - 3603
chroot	647	1353 - 1530	2433 - 2563	2228 - 2228	2228 - 2228	2467 - 2559	3923 - 4078	3934 - 4070
cut	635	1279 - 1455	2317 - 2467	2118 - 2118	2118 - 2118	2363 - 2435	3824 - 3950	3784 - 4017
dircolors	559	1228 - 1385	2190 - 2340	1967 - 1967	1967 - 1967	2244 - 2328	3494 - 3653	3511 - 3690
expand	524	1177 - 1350	2119 - 2221	1860 - 1860	1860 - 1860	2159 - 2201	3328 - 3461	3343 - 3512
fold	523	1186 - 1328	2084 - 2238	1892 - 1892	1892 - 1892	2136 - 2194	3328 - 3480	3347 - 3514
install	2108	2883 - 3219	5605 - 5874	5227 - 5227	5227 - 5227	5621 - 5836	9393 - 9789	9432 - 9788
logname	438	1089 - 1191	1899 - 2027	1661 - 1661	1661 - 1661	1939 - 2011	2959 - 3085	2969 - 3115
mknod	558	1258 - 1443	2208 - 2378	2042 - 2042	2042 - 2042	2254 - 2316	3560 - 3710	3587 - 3761
nohup	509	1168 - 1315	2038 - 2184	1817 - 1817	1817 - 1817	2100 - 2148	3212 - 3338	3225 - 3378
pathchk	471	1132 - 1272	1996 - 2134	1762 - 1762	1762 - 1762	2042 - 2104	3166 - 3269	3155 - 3304
ptx	3751	4036 - 4137	8403 - 8734	8053 - 8053	8053 - 8053	8510 - 8640	14899 - 15261	14759 - 15002
rmdir	490	1149 - 1297	2019 - 2141	1782 - 1782	1782 - 1782	2073 - 2113	3180 - 3283	3158 - 3322
sha256sum	682	1312 - 1560	2396 - 2580	2266 - 2266	2266 - 2266	2460 - 2548	4033 - 4184	4055 - 4254
sleep	485	1142 - 1272	2018 - 2141	1768 - 1768	1768 - 1768	2044 - 2117	3156 - 3268	3147 - 3312

stty	858	1456 - 1852	2634 - 2818	2516 - 2516	2516 - 2516	2704 - 2786	4428 - 4612	4444 - 4607
tee	531	1185 - 1307	2085 - 2219	1851 - 1851	1851 - 1851	2139 - 2189	3290 - 3431	3302 - 3459
TRUE	429	1077 - 1197	1867 - 2005	1628 - 1628	1628 - 1628	1919 - 1981	2914 - 3022	2918 - 3078
unexpand	541	1194 - 1332	2148 - 2254	1892 - 1892	1892 - 1892	2186 - 2244	3407 - 3544	3387 - 3586
vdir	2909	3305 - 4087	6669 - 6849	6272 - 6272	6272 - 6272	7369 - 8058	11402 - 11658	11342 - 11742
base32	632	1555 - 1652	2747 - 2907	2359 - 2359	2359 - 2359	2793 - 2855	4221 - 4411	4240 - 4394
chcon	1195	1885 - 2130	3579 - 3750	3308 - 3308	3308 - 3308	3600 - 3762	5935 - 6207	5980 - 6172
cksum	1119	1836 - 2125	4102 - 4384	4311 - 4311	4311 - 4311	4206 - 4326	7812 - 8198	7869 - 8235
date	1513	4969 - 5148	9039 - 10324	7364 - 7364	7364 - 7364	10707 - 12341	13671 - 13974	13519 - 13790
dirname	454	1098 - 1221	1918 - 2056	1693 - 1693	1693 - 1693	1982 - 2012	3036 - 3149	3030 - 3186
expr	3243	3612 - 3818	7493 - 7786	7060 - 7060	7060 - 7060	7576 - 7700	13091 - 13450	12959 - 13297
groups	504	1163 - 1321	2053 - 2213	1813 - 1813	1813 - 1813	2105 - 2187	3238 - 3368	3232 - 3394
join	884	1463 - 1644	2633 - 2831	2514 - 2514	2514 - 2514	2691 - 2799	4425 - 4599	4421 - 4622
ls	2909	3305 - 4087	6669 - 6849	6272 - 6272	6272 - 6272	7369 - 8058	11402 - 11658	11342 - 11742
mktemp	522	1170 - 1342	2062 - 2202	1826 - 1826	1826 - 1826	2122 - 2170	3246 - 3417	3243 - 3408
nproc	515	1174 - 1311	2070 - 2228	1870 - 1870	1870 - 1870	2120 - 2196	3300 - 3432	3327 - 3452
pinky	561	1492 - 1644	2616 - 2754	2201 - 2201	2201 - 2201	2638 - 2740	3979 - 4105	3969 - 4150
pwd	495	1151 - 1281	2007 - 2167	1772 - 1772	1772 - 1772	2033 - 2125	3165 - 3276	3161 - 3321

runcon	447	1111 - 1292	1944 - 2074	1695 - 1695	1695 - 1695	1994 - 2046	3034 - 3181	3032 - 3196
sha384sum	682	1312 - 1559	2432 - 2594	2298 - 2298	2298 - 2298	2508 - 2580	4107 - 4260	4112 - 4300
sort	2195	2591 - 2903	5214 - 5447	5037 - 5037	5037 - 5037	5265 - 5357	9174 - 9380	9079 - 9354
sum	673	1435 - 1653	2625 - 2771	2419 - 2419	2419 - 2419	2659 - 2743	4271 - 4437	4285 - 4456
test	629	1332 - 1426	2406 - 2554	2156 - 2156	2156 - 2156	2452 - 2526	3923 - 4014	3872 - 4097
truncate	574	1516 - 2007	2652 - 2794	2268 - 2268	2268 - 2268	2662 - 2764	4071 - 4195	4023 - 4193
uniq	655	1302 - 1404	2308 - 2488	2126 - 2126	2126 - 2126	2384 - 2422	3767 - 3959	3726 - 3944
wc	708	1393 - 1551	2553 - 2695	2280 - 2280	2280 - 2280	2567 - 2657	4138 - 4247	4098 - 4348
base64	638	1536 - 1640	2680 - 2856	2323 - 2323	2323 - 2323	2726 - 2838	4191 - 4330	4166 - 4382
chgrp	1225	1870 - 2071	3546 - 3721	3331 - 3331	3331 - 3331	3579 - 3717	5840 - 6217	5951 - 6160
comm	568	1221 - 1384	2139 - 2339	1942 - 1942	1942 - 1942	2233 - 2283	3473 - 3604	3465 - 3648
dd	1113	1720 - 1819	3208 - 3378	3005 - 3005	3005 - 3005	3288 - 3418	5424 - 5566	5406 - 5581
du	4568	5245 - 5355	10848 - 12332	10378 - 10378	10378 - 10378	11684 - 12919	19153 - 19478	18931 - 19333
factor	1227	1679 - 1855	3271 - 3503	3149 - 3149	3149 - 3149	3379 - 3591	5810 - 5924	5690 - 5869
head	640	1331 - 1758	2425 - 2559	2209 - 2209	2209 - 2209	2451 - 2511	3924 - 4048	3874 - 4083
kill	534	1166 - 1317	2068 - 2202	1840 - 1840	1840 - 1840	2108 - 2190	3305 - 3419	3306 - 3456
md5sum	669	1302 - 1547	2394 - 2586	2247 - 2247	2247 - 2247	2448 - 2532	4030 - 4148	4013 - 4181
mv	2293	3047 - 3527	6016 - 6196	5576 - 5576	5576 - 5576	6028 - 6218	10136 - 10514	10144 - 10542

numfmt	919	1578 - 1968	2897 - 3129	2782 - 2782	2782 - 2782	2980 - 3033	4943 - 5104	4952 - 5091
pr	1596	3210 - 3334	5830 - 6092	5125 - 5125	5125 - 5125	6596 - 7221	9235 - 9413	9241 - 9507
readlink	789	1386 - 1589	2563 - 2712	2306 - 2306	2306 - 2306	2545 - 2692	4202 - 4368	4151 - 4368
seq	590	1249 - 1403	2205 - 2365	1974 - 1974	1974 - 1974	2273 - 2330	3571 - 3700	3573 - 3713
sha512sum	682	1312 - 1555	2424 - 2594	2298 - 2298	2298 - 2298	2504 - 2566	4143 - 4272	4124 - 4307
split	869	1577 - 1673	2935 - 3089	2725 - 2725	2725 - 2725	2985 - 3055	4877 - 5063	4856 - 5012
sync	476	1142 - 1280	1998 - 2148	1770 - 1770	1770 - 1770	2034 - 2104	3120 - 3289	3147 - 3303
timeout	576	1228 - 1367	2145 - 2302	1918 - 1918	1918 - 1918	2205 - 2273	3423 - 3540	3420 - 3578
tsort	545	1204 - 1354	2168 - 2292	1919 - 1919	1919 - 1919	2184 - 2274	3438 - 3596	3404 - 3570
unlink	437	1088 - 1197	1900 - 2026	1662 - 1662	1662 - 1662	1936 - 2006	2959 - 3079	2947 - 3128
who	582	1231 - 1375	2177 - 2341	1941 - 1941	1941 - 1941	2211 - 2305	3478 - 3606	3448 - 3619
basename	460	1108 - 1203	1946 - 2070	1713 - 1713	1713 - 1713	1998 - 2048	3081 - 3183	3053 - 3234
chmod	1195	1817 - 2056	3475 - 3627	3250 - 3250	3250 - 3250	3462 - 3608	5780 - 6034	5790 - 6077
cp	1925	2563 - 3162	5101 - 5293	4750 - 4750	4750 - 4750	5147 - 5382	8607 - 8920	8583 - 8926
df	1555	2116 - 2455	4020 - 4208	3783 - 3783	3783 - 3783	4028 - 4240	6820 - 7079	6840 - 7037
echo	501	1135 - 1315	1977 - 2131	1775 - 1775	1775 - 1775	2029 - 2093	3154 - 3269	3160 - 3287
FALSE	429	1077 - 1200	1881 - 2015	1628 - 1628	1628 - 1628	1917 - 1981	2915 - 3036	2913 - 3080
hostid	437	1087 - 1187	1899 - 2029	1658 - 1658	1658 - 1658	1933 - 1997	2959 - 3076	2962 - 3111

link	438	1089 - 1207	1905 - 2033	1665 - 1665	1665 - 1665	1931 - 2009	2968 - 3100	2972 - 3127
mkdir	680	1388 - 1521	2484 - 2658	2284 - 2284	2284 - 2284	2524 - 2592	4053 - 4198	4064 - 4226
nice	519	1153 - 1280	2015 - 2165	1817 - 1817	1817 - 1817	2069 - 2127	3213 - 3381	3199 - 3370
od	869	1503 - 1731	2777 - 2932	2663 - 2663	2663 - 2663	2840 - 2937	4697 - 4844	4672 - 4857
printenv	444	1091 - 1204	1919 - 2053	1673 - 1673	1673 - 1673	1947 - 2015	2990 - 3128	3000 - 3156
realpath	838	1425 - 1547	2650 - 2783	2380 - 2380	2380 - 2380	2616 - 2775	4327 - 4483	4291 - 4514
sha1sum	668	1303 - 1561	2417 - 2583	2283 - 2283	2283 - 2283	2473 - 2555	4054 - 4251	4084 - 4299
shred	878	1588 - 1786	2950 - 3126	2709 - 2709	2709 - 2709	2984 - 3086	4840 - 5004	4852 - 4978
stat	2145	2352 - 2496	4334 - 4546	4216 - 4216	4216 - 4216	5102 - 5783	7322 - 7522	7349 - 7533
tac	3066	3499 - 3656	7202 - 7497	6765 - 6765	6765 - 6765	7280 - 7446	12564 - 12857	12410 - 12721
touch	1297	4440 - 4599	7934 - 8134	6286 - 6286	6286 - 6286	8604 - 9337	11698 - 11886	11579 - 11782
tty	435	1085 - 1192	1891 - 2023	1656 - 1656	1656 - 1656	1933 - 2007	2969 - 3066	2945 - 3116
uptime	807	1768 - 1880	3314 - 4649	3000 - 3000	3000 - 3000	4084 - 5321	5383 - 5654	5380 - 5647
whoami	441	1092 - 1197	1902 - 2034	1666 - 1666	1666 - 1666	1942 - 2012	2967 - 3098	2962 - 3119

VITA

David Reyes earned his Bachelor of Science in Computer Science from The University of Texas at El Paso in the fall of 2014. He completed the Master of Science in Software Engineering program with a concentration in Secure Cyber-Systems in 2016. After graduating with a Master of Science in Software Engineering, David joined the Ph.D. program in 2017 under the guidance of Dr. Salamah Salamah and Dr. Jaime Acosta.

David has six years of internship experience; through those six years, he has worked with the City of El Paso Department of Technology Information Services, Lockheed Martin Corporation, MIT Lincoln Laboratory, Raytheon, and Sandia National Laboratories.

At the University of Texas at El Paso, David participated in the CyberRIG Lab, where he focused on work related to software diversification and assisted in the development of challenges for the CyberRIG 2022 annual hackathon. He also served as a teaching assistant for the Computer Science Department for multiple Computer Science courses, including software reverse engineering and software engineering I and II.

David has received numerous honors and awards, including the Murchison Graduate Scholarship, National GEM Consortium Fellowship, Google-CAHSI Dissertation, Faculty Start-Up Award, and the NSF CyberCorps Scholarship for Service Fellowship.

This dissertation was typed by David Reyes.