ENFORCING CONTROL-FLOW INTEGRITY IN FLEXOS

A REPORT SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF BACHELOR OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2023

Mateusz Krajewski

Project Supervisor: Pierre Olivier

Department of Computer Science

Contents

Word Count: 10280

List of Tables

List of Figures

Abstract

ENFORCING CONTROL-FLOW INTEGRITY IN FLEXOS
Mateusz Krajewski
A report submitted to The University of Manchester
for the degree of Bachelor of Science, 2023

FlexOS is an open-source operating system focused on enforcing security through isolation. It divides program libraries across several memory domains so that when one is compromised, the others remain intact. Recently, a vulnerability has been discovered in FlexOS that makes it possible to execute code from anywhere in the system, as long as it does not access any memory. It is an unintended violation of control-flow integrity and might serve as a basis for certain families of code-reuse attacks. This report investigated the impact of said vulnerability and explored several approaches to mitigating it. The proposed mitigation techniques were less efficient than expected, but they shed more light on the complexity of the problem and laid solid groundwork for further research.

Declaration

No portion of the work referred to in this report has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

Copyright

Acknowledgements

The author would like to thank Pierre Olivier and Hugo Lefeuvre for their patience and support throughout the year.

Chapter 1

Introduction

In the era of constant technological advancement and the prevalence of computer systems in all industries and all aspects of life, system security has become a significant point of interest. This area of science features an endless race where antagonistic forces keep discovering new vulnerabilities. At the same time, researchers work on eliminating those, or at least making them so difficult to exploit that it is no longer viable to try. The size of the global cyber security market is expected to grow from $200 billion to $900 billion in the next ten years [58]. Technological advancement in cybercrime never ends, and neither does its defensive counterpart.

As a part of the ongoing study on computer system security, a new operating system has been developed – FlexOS [33]. Since it is still a relatively recent addition to the research efforts, the system features a vast potential for improvement. In particular, it has exhibited a flaw that makes it vulnerable to certain families of attacks – known as Return- and Jump-Oriented Programming. This report investigates the flaw, examines its potential impact on the security of FlexOS, and explores possible mitigation techniques.

1.1 Context

Isolation

Isolation is one of the many ways software can be protected from potential attacks: sensitive data might be isolated from the rest of the software, so it is not leaked even in the case of a severe security breach, or a program can be divided into multiple parts isolated from one another, so even if one is compromised, the others remain safe. There have been numerous attempts at providing

isolation-based security with implementations that could incur the lowest possible cost on the system: ERIM [56] was a hardware-enforced protection technique that could be incorporated into any application effortlessly and with negligible performance overhead; Hodor [25] introduced the idea of user-space 'protected libraries' for services that would otherwise require kernel-level permissions. Those libraries offered equal protection and performance while reducing the overall load on the kernel. FlexOS [33] made it possible to isolate program libraries and quickly switch between different isolation implementations.

When creating new software, developers must always decide whether and how to apply an isolation strategy. Different strategies yield different benefits and drawbacks – some are better suited against certain types of attacks, others may be less so, but the range of their use cases is broader. The ultimate choice always depends on the purpose of the software, the data it handles, and multiple other factors. Most often, that choice must also be made at design time. Making that decision later in the development process, e.g., in response to a vulnerability found in the currently used strategy, incurs a significant refactoring effort. What if it was different, however? What if, upon discovering that a chosen isolation strategy has been compromised, the developers could easily change it at any time? FlexOS addresses this question.

FlexOS

FlexOS is an open-source operating system that strongly focuses on enforcing proper program control flow. To achieve that, it makes use of the idea of *compartmentalisation* – the user is encouraged to divide program libraries among several *compartments*, or *memory domains*, e.g., with the help of the SOAAP compartmentalisation framework [24]. These can communicate with one another only through strictly guarded points called *gates*. Compartmentalisation in FlexOS drastically reduces the cost of switching between different security strategies [34].

FlexOS supports multiple implementation back-ends; this report focuses on Intel Memory Protection Keys, a hardware mechanism intended to facilitate page-based memory protection in Intel CPUs. There are 16 keys available in total, and every memory page can be assigned to a key; that information is stored in a 32-bit register `PKRU`. Changing access permissions for a set of pages assigned to a single key is as easy as modifying the permissions of that key [16].

To befit the purposes of the following discussion, the implementation of FlexOS has been modified to allow for compatibility with the `clang` compiler.

1.2 Motivation

Intel MPKs exhibit a limitation that affects FlexOS and makes it potentially vulnerable to attacks – they can only specify whether a given key has a read or write memory access [14]; they do not account for execution permissions, and, by extension, neither do FlexOS compartments. Suppose a potential attacker gained control of a memory domain and attempted to access memory in a different, non-compromised domain. In that case, the system would crash – FlexOS and the underlying MPK logic would account for that. However, if the attacker tried to execute a function that does not access memory at all, they would succeed regardless of the compartment in which that function is located. Currently, the system does not provide any protection against unauthorised code execution. This report aims to explore whether these kinds of exploits are viable at all in FlexOS and, if they are, what it takes to prevent them.

FlexOS stands at the forefront of modern research into system security. Detecting, investigating, and mitigating its vulnerabilities is a direct contribution to that research and a step toward developing robust, secure, and reliable software.

1.3 Research questions

In order to accurately address the problem, two research questions were formulated:

1. What are the possible ways of mitigating the vulnerability in FlexOS?
2. How effectively do the proposed preventive measures fulfill their purpose?

Chapter 2 gives some background necessary to understand the topic at hand and provides a brief outline of the current research landscape. Research Question 1 is answered in Chapter 3, which proposes several possible methods of mitigating the vulnerability; it also lays the groundwork for answering Research Question 2 by describing the methodology of testing their effectiveness. Chapter 4 answers RQ2 by providing a detailed breakdown of the experiment results. After establishing the severity of the vulnerability, Chapter 5 describes several possible ways of exploiting it. Chapter 6 concludes the experiment by acknowledging its outcomes and suggesting directions for further research.

Chapter 2

Background

This chapter discusses several topics necessary to understand the problem in depth.

2.1 Control Flow Integrity

A typical program has a specific control flow that depends on its inputs. Every instruction is executed in sequence, except in places where function or system calls are made or where a branch instruction is encountered. These places are particularly vulnerable – a potential attacker might try to exploit them to subvert the control flow and trigger some malicious behaviour. Enforcing Control Flow Integrity refers to techniques that endeavour to prevent that [4].

Two kinds of such attacks are explored in detail in this report – Return- and Jump-Oriented Programming. It is suspected that those attacks, coupled with the existing vulnerability in FlexOS (refer to Section 1.2), could potentially instigate a severe security breach.

2.2 Return-Oriented Programming

A type of attack that allows a malicious agent to subvert the program's control flow. By gaining control of the stack, the potential attacker could chain together short instruction sequences, each ending with the `ret` instruction – those sequences are called *gadgets* – thus allowing for arbitrary behaviour to be executed [45]. This kind of attack may be deadly – there is no need to inject any code since the entire necessary framework is already there. That makes it immune to many mitigation techniques protecting against code injection, like W$\oplus$X [20], ASLR [52], or DEP [36].

The attack has been proven to simulate Turing-complete behaviour [45], even when gadgets are few and of poor quality [19]. However, so far, no one has attempted to chain gadgets that do not access memory; if possible, it could serve as a basis for exploiting the vulnerability in FlexOS (described in Section 1.2). To protect against that, one may attempt to limit the number of such gadgets, e.g., by forcing memory access before every `ret` instruction. This report explores this idea further.

2.3 Jump-Oriented Programming

This attack is equivalent to ROP. It does not rely on a program stack, instead making use of what is referred to as a *dispatcher gadget*. The gadgets that serve as 'building blocks' of the attack end with indirect branch instructions – unlike in ROP, where they end with `ret` [9]. The general principles of the attack remain the same – gadgets can be chained together to cause undesired behaviour. Chaining *vulnerable gadgets* (i.e., those that do not access memory) in FlexOS is potentially dangerous due to the existing vulnerability and could supposedly be prevented by forcing memory access before every indirect branch instruction.

2.4 LLVM

LLVM is a compiler infrastructure suited for low-level code manipulation and analysis. It includes a C/C++ compiler `clang`, which is compatible with `gcc` but offers many additional tools and functionalities for researchers and developers. The compilation process includes several compiler passes, which first modify the initial *Intermediate Representation (IR)*; this is then turned into assembly code and later into binary code (see Figure 2.1). It is possible to create one's own LLVM pass and incorporate it into the compilation process [46]. This functionality will be explored further, in the attempts to prevent potential ROP and JOP attacks in FlexOS.

LLVM IR breaks down a program into *modules*; these consist of *functions*, which consist of *basic blocks* (see Figure 2.2) – those are going to be of particular interest here. According to the LLVM Language Reference Manual, a basic block comprises a sequence of non-terminating instructions ended with a terminating instruction (like `ret` or a branch) [32]. It is possible to create an LLVM pass that would force memory access before every `ret` instruction (in the context of ROP), every branch instruction (in the context of JOP), or every terminating instruction (to ensure maximum security). A program compiled with such a pass could potentially feature more robust protection against ROP and

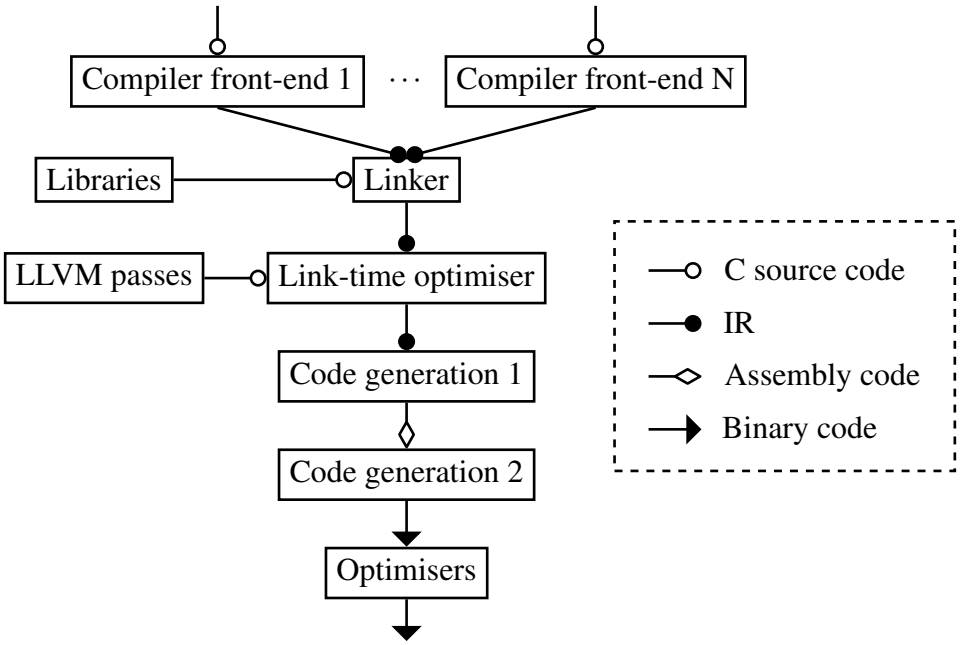

Figure 2.1: LLVM compilation process [32].

JOP attacks in FlexOS.

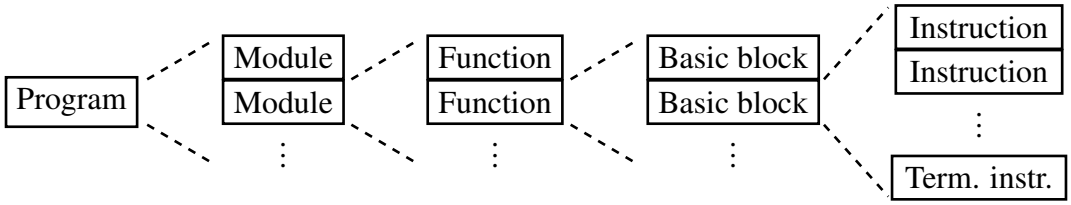

Figure 2.2: LLVM IR hierarchy [32]. A program consists of modules; a module consists of functions, each of which consists of basic blocks. A basic block is a sequence of instructions ending with a terminating instruction.

2.5 Related work

A fair amount of research has already been done on isolation, compartmentalisation, and code-reuse attacks, like Return- or Jump-Oriented Programming.

Isolation

The idea of isolating parts of the program to increase its overall security and resilience to attacks has been known for a long time. Over the years, researchers have developed tools that facilitate the use of isolation in real-world applications, like SOAAP [24] – which makes it easy to reason about compartmentalisation at design time – or PtrSplit [38] – meant to assist with pointer management during the partitioning process.

There have been numerous attempts at creating software that could aid with implementing isolation most efficiently. Many of them have made use of Intel MPKs, just like FlexOS, and focused on isolating program libraries from one another: Hodor [25] introduced the idea of protected libraries in order to reduce the computational load on the kernel; CubicleOS [48] divided a program into 'cubicles,' but at the cost of performance; Wedge [8] successfully implemented fine-grained and efficient program compartmentalisation; Cali [7] introduced the idea of compiler-assisted isolation; UnderBridge [23] isolated critical system services from the rest of the system. Donky [50] additionally proposed a new implementation of domain keys, which was proven to be both faster and more secure than Intel MPKs. Aside from isolating program libraries from one another, significant effort was put into isolating trusted from untrusted code as well: ERIM [56], ConfLLVM [10], Privtrans [12], and many others are living examples of that.

Intel additionally developed a mechanism called Software Guard Extensions (SGX), which made it possible to define small regions of memory – enclaves – where program execution was protected from malicious activities [26, 40]. Glamdring [37] and EActors [47] further built on this idea by automating the isolation process.

The mentioned software, however, does not emphasise easy switching between different implementation back-ends, which is specific to FlexOS and makes it exceptionally flexible and secure.

ROP and JOP attacks

Code-reuse attacks, Return- and Jump-Oriented Programming in particular, have been the focus of intense research for many years. These kinds of attacks keep evolving – recently, several new techniques of disguising ROP gadgets have been developed, which circumvent many of the already known defences [13]. Additionally, it was proven that even with little helpful code, minimal assumptions, and strict code-reuse protections, it might be possible to perform a successful attack [19, 22, 57]. Some tools can do it automatically [57]. With the potential attacks posing such a threat, the developers need to be able to assess the state of their current CFI defences – there are tools for that, as well

[41].

When defending against ROP attacks alone, shadow stacks are one of the most popular approaches – they protect return addresses from tampering by introducing an additional stack that contains only those [17]. A large amount of research was devoted to improving the idea: by the use of cryptographic codes [39], restricting returns only to 'active' call sites [18], or splitting the stack in two in order to isolate instruction pointers [31]. Shadow stacks are not the only way, however – there were proposals to discard return instructions altogether and replace them with safer alternatives [35].

As for protecting against JOP attacks, the most popular basis for the research is indirect branch tracking, i.e., ensuring that every indirect branch leads only to a valid target address. This method has been improved over the years by restricting sensitive function calls only to specific places in the program [51], embedding instruction IDs in order to validate branch targets [4], using jump tables to keep track of branches [55], or using a bitmap to verify whether there is a valid branch target in given places in the code [6]. NSA additionally proposed an extension to the existing instruction sets with two new instructions – a jump landing point and a call landing point [42].

Several other defence strategies were investigated – Intel developed an ISA extension that made use of both shadow stacks and indirect branch tracking [53]; there were proposals of hardware-based branch regulation, protecting against jumping into the middle of a function [30], and binary recompilation, meant to produce binaries with fewer and worse-quality gadgets [11]. However, research on the above has yet to be done in the context of FlexOS. Similarly, this is the first time the vulnerability described in Section 1.2 is being investigated.

Chapter 3

Defense design and implementation

In order to answer Research Question 1, four approaches to mitigating the vulnerability were designed. Their detailed descriptions can be found in Section 3.1. Three programs were subject to the analysis; Section 3.2 outlines the reasoning behind the choice of those programs and their potential contribution to answering Research Question 2. Section 3.3 describes the metrics used to benchmark the chosen programs, and Section 3.4 sums up the overall workflow of the experiment.

The research explored various directions, many of which were ultimately abandoned. Section 3.5 briefly summarises the most significant of those and the reasons behind their ultimate withdrawal from the research.

3.1 Program variants

Pursuant to the discussion on Research Question 1, four approaches to mitigating the vulnerability were proposed. Two of them, described in Sections 3.1.2 and 3.1.3, focused on modifying C source code, whereas the other two, outlined in Sections 3.1.4 and 3.1.5, modified LLVM IR. The first three aimed at minimising the number of vulnerable ROP gadgets by protecting function exit points; the last one protected all basic blocks in order to prevent both ROP and JOP attacks.

What is meant by a 'minimal memory access'

Sections 3.1.2 to 3.1.5 describe program variants that performed what is referred to as *minimal memory access* in certain places in the program. The reason for doing so at all was to prevent unauthorised

function execution, as outlined in Section 1.2. Such action had to be as simple as possible to minimise its impact on the overall performance. Given a variable `VAR`, that could be achieved by so much as `VAR=0;` in C, or `store volatile i32 0, i32* @VAR` in LLVM IR. In order for this scheme to work, such a variable had to be:

1. global – accessing a local variable made no sense in this context because those are stored on the program stack. Suppose the attacker had full control of the stack, as is the case in Return-Oriented Programming. In that case, the memory access would not be considered 'unauthorised' because the variable would be located in the memory region under the attacker's control. That being said, even using global variables was not fully correct – that is further explained in Section 4.3. The equivalent of a global variable in LLVM IR is called an *internal* variable.

2. volatile – any optimiser would have considered such memory access redundant and attempted to remove it altogether. Declaring a variable as volatile prevented it from being 'optimised away' in the compilation process.

3.1.1 Variant 1 – plain

It was a basic version of the program, without any modifications.

3.1.2 Variant 2 – instrumentation calls

This variant used the `-finstrument-functions` compiler flag. The flag forced the program to perform what is called an *instrumentation call* at every function entry point and every function exit point – though in this case, only the latter was relevant. The instrumentation call was made to a special custom-made function, which performed minimal memory access.

3.1.3 Variant 3 – exit points at source level

In this case, minimal memory access was injected at every function exit point at the source code level. Recognising exit points was a matter of more than simple lexical parsing; some knowledge of the C language syntax was also necessary. For that purpose, the tool called *Coccinelle* [5] was utilised. Coccinelle was designed to perform advanced source-to-source transformations on C source code; it could easily recognise exit points and apply the desired changes. By applying the protection, it could possibly prevent ROP attacks.

3.1.4 Variant 4 – exit points at IR level

This program variant injected minimal memory access at every function exit point – denoted by a `ret` instruction – at the IR level. During compilation in `clang` (Figure 2.1), several diagnostic and optimising passes are performed on the IR; it is possible to create one's own pass and include it in the process [46]. The pass created for the experiment located all basic blocks ending with a `ret` instruction and applied the desired change to them. Similar to the above, it could possibly prevent ROP attacks.

3.1.5 Variant 5 – basic blocks

Here, minimal memory access was injected at the end of every basic block. The method used was very similar to that mentioned in Section 3.1.4 – this time, however, all basic blocks were modified. That dramatically increased the protection granularity. Theoretically, it would have been enough to protect exit points and indirect branches to prevent ROP and JOP attacks. This approach, however, took a safer route and protected all possible basic blocks.

3.2 Tested programs

The experiment focused on analysing three programs – a 'Hello World,' a simple program with many functions, and Redis, an in-memory data store. The following sections outline the reasoning behind the choice of those programs, the metrics that were the most relevant to their analysis, and the benchmarks used to measure them.

3.2.1 A 'Hello World' program

The program consisted of only nine lines of code and did nothing more but print a 'Hello World' to the standard output; its listing can be found in Figure 3.1. The conditional statement was necessary to ensure that more than two basic blocks ended with `ret` (one for `main` and one for `print_hw` functions). Without that statement, there would have been no difference between Variant 4 (see Section 3.1.4) and Variant 5 (see Section 3.1.5) of the program. As such, there were six basic blocks in total.

Due to the program's simplicity, it could not be measured for performance; any changes to the running time caused by applying mitigation techniques would have been minuscule and yielded little

significant information. However, that simplicity also made it possible to easily observe the program's behaviour. The changes in the number of assembly code lines and the number of gadgets promised to offer insight into how the proposed preventive measures affected small, simple software; that could be further used to derive findings for longer and more complicated programs.

```
#include <stdio.h>
#include <string.h>

void print_hw() {
        printf("Hello world!\n");
}

int main(int argc, char **argv) {
        if(argc == 1)
                print_hw();
}
```

Figure 3.1: A simple 'Hello World' program

3.2.2 A simple program with many functions

This program consisted of 10,000 short functions containing a mix of loops and conditional statements; it was intended to minimise the ratio of return blocks to all basic blocks and explore the impact of the proposed changes in such circumstances. Indeed, that ratio reached almost 1:105 in the tested program. Here, the effect on code size, particularly when protections were applied to all basic blocks, was much more pronounced. Additionally, gadget analysis could shed light on the relationship between code size and the number of gadgets. It would have been, however, unreasonable to measure the performance of this program – due to its speculative nature and the lack of practical uses, a discussion on its running time would have been meaningless.

3.2.3 Redis

Redis is a popular open-source in-memory data store. It is a massive code base with over 170,000 lines of code and hundreds of contributors. Due to its popularity, size, and the fact that it is already integrated with FlexOS, Redis was a perfect example of how the proposed preventive measures could affect a real-world application.

The experiment tested Redis Version 5.0.6, modified to add compatibility with FlexOS. The modifications included dividing its dependent libraries across three compartments. Redis was run and benchmarked in a Docker (Version 23.0.1) container running a FlexOS virtual machine.

3.3 Metrics

The programs were tested on three different metrics: performance, the static assembly instruction count, and the overall number of ROP and JOP gadgets. The reasoning behind these metrics and their evaluation methods are described in detail in the following sections.

3.3.1 Performance

Performance was arguably the most important of the three proposed metrics; it indicated the impact of applying mitigation techniques on the overall program operation. If a modification induced a significant drop in performance, it might not have been viable to use it in practical applications. Further, the proposed preventive measures could have had a different impact on programs of different sizes and complexity.

Due to the varying nature of the tested software, it was not viable to measure performance for simple programs. The 'Hello World' was too small to exhibit significant changes in running time. Similarly, the simple program with many functions was too speculative for the results to be meaningful in any way – it was expected to yield more valuable insight during the assembly code and gadget analysis. Therefore, only Redis was tested in terms of performance.

The Redis benchmark was done with the use of an already existing tool [44]. The program sent a considerable amount of GET and SET requests to the Redis server and recorded the number of those that were successful; the results were then divided by the type and the size of the request and were represented as the number of requests per second.

3.3.2 Assembly code

This metric gave a general idea of how code size evolved after being subject to the proposed CFI enforcement techniques. Generally, code size is less reliable than performance since it does not necessarily have a significant impact on the running time [27]. However, this report focused on a very particular situation – if a program function was executed in its plain version, then any modifications

applied right before its exit points were bound to be executed as well. Hence, if two different techniques resulted in the addition of different numbers of assembly code lines, the resulting performance was guaranteed to be different – the more LoC added, the worse the performance.

Assembly code analysis is usually more important for embedded system developers, who need to put particular care into minimising the size of the developed code and its loading time [28]. However, even in systems that are not inherently embedded, like FlexOS, it is an important indicator of when the effects of code modifications start spiralling out of control.

Extracting assembly code with `clang` was a two-step process: first, LLVM IR was generated using the `-emit-llvm` compiler flag. Then the IR was compiled into assembly code using `llc`, an LLVM static compiler [3]. Further analysis was performed using standard Unix tools, like `wc` [1].

3.3.3 Gadgets

Gadget analysis indicated how vulnerable the program was to hypothetical ROP and JOP attacks. The more gadgets were available, the more likely it was to create potentially useful gadget chains. Gadgets that did not access memory were of particular interest here – since FlexOS, in its current form, had no way to prevent their use. Ideally, applying minimal memory access to every exit point would have drastically reduced the number of vulnerable (i.e., those that did not access memory) ROP gadgets, and applying it to the end of every basic block would have decreased the number of vulnerable JOP gadgets. Due to their highest granularity, only variants 4 and 5 were subject to the analysis. Three attempts were made in search of the most reliable approach to performing gadget analysis. The following discussion describes them in detail and explains why only the last one was successful.

Attempt 1 – IR parsing

This attempt focused on extracting human-readable LLVM IR using the `-emit-llvm` compiler flag and performing a lexical parsing on the resulting files in search of potential gadgets. It failed due to a misconception – ROP and JOP gadgets can only be found in assembly code, which LLVM IR is yet to be assembled into; the IR itself has little to do with gadgets. This fact was also relevant in the subsequent discussion on the results.

Attempt 2 – assembly code parsing

Another attempt concentrated on performing lexical parsing on the assembly code (extracted using methods mentioned in Section 3.3.2). This approach failed for two reasons: firstly, it would have taken a profound analysis to capture all instruction variants existing in the x86-64 architecture. It failed to adopt that depth. Secondly, ROP and JOP gadgets can start in the middle of the instruction; no lexical-based parsing method could account for that. The most viable approach, it seemed, would have been to perform the analysis on binary files – this is what the last attempt focused on.

Attempt 3 – Ropper

This attempt used Ropper [49], a gadget parsing and analysis tool, to find all available gadgets. Ropper operates on binary files; hence its effectiveness in recognising potential ROP and JOP gadgets is much higher than that of lexical parsing. In particular, the tool makes it possible to locate gadgets that do not access memory and attempts to create ROP and JOP chains that simulate arbitrary operations.

All three tested programs were processed by Ropper in order to count the vulnerable gadgets, i.e., those that did not access memory. Additionally, an attempt was made to chain them and simulate the operation `execve("/bin/sh", NULL, NULL)`. If such an attempt had been successful, the attacker could spawn a shell by themselves, greatly expanding the range of malicious activities they would be capable of. Redis was the only program tested for such a vulnerability due to the number and variability of its gadgets.

3.4 Workflow

Figure 3.2 describes the workflow followed throughout the experiment. The analysis began with a basic version of the program. That was then modified to simulate the application of four different mitigation techniques, giving a total of five program variants. The techniques differed by the granularity level (source or IR) and the approach used. After the programs were compiled and run, the main benchmark measured their performance and recorded the results. Then, assembly code was extracted using the existing LLVM tools, the total number of lines of code was counted, and the results were recorded and tabulated. The program binaries were further extracted and processed by Ropper, a gadget analysis tool [49], which calculated the total number of gadgets, dividing them by their type (ROP or JOP) and whether they accessed memory or not. An attempt to create a potentially useful gadget chain was made.

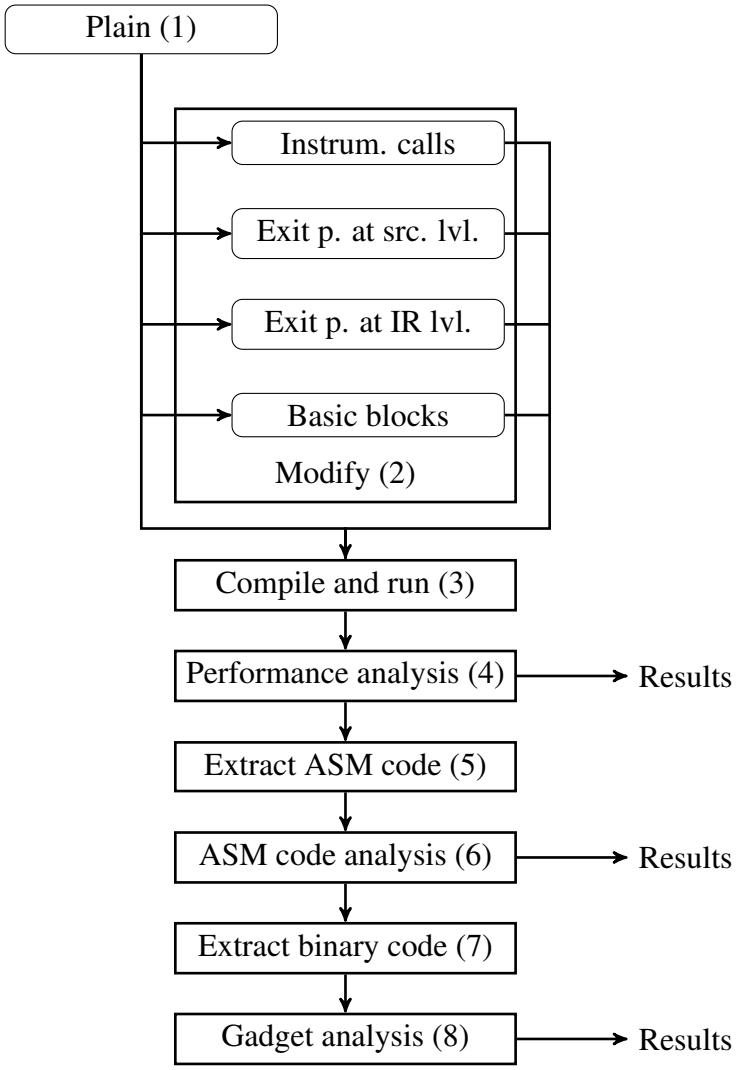

Figure 3.2: Experiment workflow. The analysis began with a basic version of the program (1). That was then modified (2) to simulate the application of four different mitigation techniques. After the programs were compiled and run (3), the main benchmark measured their performance and recorded the results (4). Then, assembly code was extracted (5), the total LoC were counted, and the results were recorded and tabulated (6). The program binaries were further derived (7) and processed by a gadget analysis tool (8), which counted the total number of gadgets and made an attempt to create a potentially useful gadget chain.

3.5 Abandoned approaches

The defense design process included a large number of potential routes, many of which were concluded to be dead ends – either due to their inefficiency, unreliability, or weak relevance to the subject

at hand. Two approaches, in particular, were abandoned very late into the analysis process. The following sections describe those approaches and outline the reasons for their ultimate withdrawal from the research.

3.5.1 libmpk

Libmpk [43] is a library providing a software abstraction for Intel MPKs. The main reason MPKs are not more widely adopted is hardware limitations – due to the size of the `PKRU` register, there can only be 16 keys at a time, one of which is reserved. Libmpk overcame this limitation by abstracting the keys, significantly increasing their usability. The library also improved overall security by increasing their compatibility with Unix and mitigating certain commonly known vulnerabilities.

Initially, the project assumed that libmpk could be incorporated into FlexOS instead of regular Intel MPKs. The initial benchmarks, therefore, focused primarily on libmpk, investigating CFI enforcement on the libmpk-specific kernel provided by the paper authors. After exploring the performance in that kernel, the following steps could be to incorporate libmpk into FlexOS, apply CFI enforcement techniques, and measure the performance again. However, the code base provided by the authors was severely outdated and cumbersome; it required some particular dependency versions, and the source code still had to be amended to account for them. Additionally, the paper authors could not produce any practical examples of libmpk applications despite them being described in detail in the paper itself. That ultimately caused this direction to be abandoned. The extensive effort needed to get the kernel and libmpk to work caused a significant amount of time to be wasted.

3.5.2 Built-in CFI in clang

`clang` offers several CFI-related compiler schemes [21]. These can be incorporated into a program using compiler flag `-fsanitize=cfi` or one of its available subsets. They are intended to prevent the potential attacker's attempts to subvert the control flow, e.g., by the malicious use of casting or function calls. Initial benchmarks assumed that vulnerability mitigation could be achieved with one or more of those schemes. However, further investigation revealed that six of seven of them were relevant only to C++ files. The one remaining had little to do with the subject at hand. That, and the fact that all benchmarked programs, as well as FlexOS itself, were written in C, ultimately caused this approach to be abandoned.

Chapter 4

Evaluation

This chapter gives a detailed overview of the results of the performed experiments. Section 4.1 features a detailed breakdown of the benchmarks and their results; those are further summarised and dissected in Section 4.2, which focuses on answering Research Question 2. Over the course of the research, several simplifying assumptions had to be made – either because of the complexity involved or simply because of the lack of time. These are outlined in Section 4.3.

4.1 Experiment results

The experiment was performed on an Intel Xeon Gold 5222 processor (16 cores/32 threads, 3.9 GHz) with Intel MPK enabled. All programs were compiled on Ubuntu 20.04.5 LTS with `clang` Version 10.0.0 and a default optimisation flag (`-O0`). Any subsequent discussion on 'assembly code' refers to x86-64 assembly, which is specific to Intel Xeon processors.

4.1.1 A 'Hello World' program

Performance analysis

Performance analysis was not performed. Due to the nominal size of the program, any results related to its running time could be considered negligible.

Assembly code analysis

The static assembly instruction count for the program can be found in Table 4.1. Note that labels and assembly directives were ignored during the count.

The basic version of the program contained 26 instructions. Adding instrumentation calls more than doubled that number; there were multiple reasons for that. Firstly, the instrumentation function consisted of additional seven instructions required to retrieve arguments from the stack and perform the memory access. Before calling it, another three instructions were needed to set the arguments and perform the call itself; that was done at every possible entry and exit point. Using instrumentation calls necessitated the addition of many assembly code instructions, which had a direct and significant impact on performance, as evidenced in the subsequent performance analysis. That same behaviour could be achieved with much less effort, which already hinted at this method being largely suboptimal.

In contrast, adding memory access directly at every exit point yielded only two additional instructions – one per exit point. That instruction was `movl $0, VAR`, where `VAR` was a global variable declared with an assembly directive. It was the same for both the source- (Variant 3) and the IR-level (Variant 4) modifications. The equivalence of these two methods was further emphasised during the performance analysis for more complex programs. Adding memory access at the end of every basic block was very similar – one additional instruction per basic block yielded six in total. For programs with a large number of basic blocks, i.e., with numerous loops and conditional statements, it could potentially have a detrimental effect on performance. After all, adding even a single if-statement in the 'Hello World' program increased the number of basic blocks by four.

	No. of instructions	**Difference**
Plain	26	+0%
Instrumentation calls	58	+123%
Exit points at source-level	28	+8%
Exit points at IR-level	28	+8%
Basic blocks	32	+23%

Table 4.1: Static instruction count for the 'Hello World' program.

Gadget analysis

The total number of gadgets (Table 4.2) may initially seem surprisingly high for such a simple program. However, there is an explanation for that – in architectures with variable-sized instructions, a

gadget can start at any address and still be considered valid. The instructions it consists of may not have been intended by the programmer, which makes ROP and JOP attacks all the more dangerous. That is why, even though there were only 26 assembly code instructions in the plain version of the program, the number of ROP gadgets reached almost 90, and the number of JOP gadgets reached 35.

In the unmodified version of the 'Hello World' program, one-third of ROP gadgets and almost half of JOP gadgets did not access memory. These were considered 'vulnerable' since, by chaining them, an attacker could potentially instigate some malicious behaviour without triggering any FlexOS defences.

Injecting any additional code had an obvious consequence – the more assembly code instructions there were, the higher the potential to create new gadgets. Furthermore, adding minimal memory access at every exit point, ironically, increased the proportion of vulnerable ROP gadgets – from 33.7% to 39.6% – and did not change the number of JOP gadgets at all. Adding further protections to basic blocks availed nothing. As a reminder – protecting exit points was supposed to prevent, or at least decrease, the possibility of ROP attacks, while protecting all basic blocks was supposed to do the same for both ROP and JOP. However, the fact that gadgets could start at any address caused the results to be less predictable than expected – that observation will be touched upon again in further discussion.

An attempt was made to create a potentially useful gadget chain from the available gadgets. Due to the program size, however, this attempt yielded no meaningful results.

	ROP gadgets	**Vulnerable ROP gadgets**	**Proportion**
Plain	89	30	33.7%
Exit points at IR-level	91	36	39.6%
Basic blocks	91	36	39.6%
	JOP gadgets	**Vulnerable JOP gadgets**	**Proportion**
Plain	35	16	45.7%
Exit points at IR-level	35	16	45.7%
Basic blocks	35	16	45.7%

Table 4.2: The number of gadgets for the 'Hello World' program.

4.1.2 A simple program with many functions

Performance analysis

Performance analysis was not carried out due to the program's unrealistic nature. In this case, the focus was more on the impact on assembly code and gadgets; any discussion on performance could be considered pointless since the program did not exhibit any meaningful behaviour.

Assembly code analysis

The plain version of the program consisted of over 815,000 assembly-code instructions, excluding labels and assembly directives. Adding instrumentation calls increased that number by 12%. Just like in the 'Hello World' program, additional instructions had to be added to the entry and exit points of each of the 10,000 functions. While the increase was not as high as for some other methods described further, it could have been lowered by discarding thousands of lines of redundant code. No memory was supposed to be accessed at entry points, after all, and all such instrumentation calls were made to an empty function. That is one of the caveats of the `-finstrument-functions` flag – it forces the programs to perform both types of calls, regardless of whether they avail anything.

As was the case for the 'Hello World' program, there was little difference between applying protections to exit points at the source and the IR level. The location of the additional memory access generally translated to the same place in the assembly code for both methods. In some cases, it was coupled with putting additional data on the stack or in registers – hence the minor discrepancy. Overall, though, the number of assembly code lines grew only by about 1% after applying the protections, which could prove these methods' efficiency for these kinds of hypothetical programs.

The number of assembly code lines increased by 45.8% when the protections were applied to all basic blocks. It was not a surprising result – due to the high density of conditional statements and loops, the number of basic blocks was bound to be significantly higher than the number of return blocks alone. Moreover, since applying protection was generally equivalent to injecting a single line of code, the amount of additional code was proportionally higher.

Gadget analysis

The first observation that comes to mind when looking at Table 4.4 is how many more ROP gadgets there were compared to their JOP equivalents. The difference was massive – 66,760 ROP gadgets, as opposed to 112 JOP gadgets. This was one of the many indicators coming into sight and hinting

	No. of instructions	**Difference**
Plain	818223	+0%
Instrumentation calls	915195	+12%
Exit points at source-level	829447	+1%
Exit points at IR-level	828223	+1%
Basic blocks	1192842	+46%

Table 4.3: Static instruction count for the simple program with many functions.

that gadgets might behave contrary to the initial assumptions. In particular, their number could not be reliably estimated based on the number of basic blocks, assembly code instructions, or even conditional jump instructions alone. There were over 105,000 of those in the program, and one might have mistakenly assumed that they could be used to create JOP gadgets. Instead, when performing such an estimation, one would have had to consider that gadgets could start at any address – a significant complication, particularly in architectures with variable-sized instructions. A profound byte-by-byte analysis would have had to be performed, with little consideration for the source code. This observation already pointed at the flaw in the experiment's assumptions – they focused too much on the intended (i.e., visible) code, and this hypothetical program proved how much it might cause them to diverge from reality.

Another unanticipated observation could be made when analysing the impact of protecting all exit points in the program – applying the protection nearly doubled the proportion of vulnerable ROP gadgets, from 37.9% to 70.8%. It was an unintended and highly alarming result – the more vulnerable gadgets there were, the easier it was for the attacker to potentially exploit the vulnerability in FlexOS. The already mentioned fact that gadgets could start at any address was a direct cause of that – adding more instructions to the program, even if they were meant to protect it, increased the potential to create new gadgets, in this case – with very severe consequences.

Due to their number, applying any protections had a negligible impact on JOP gadgets.

No attempt at gadget chaining was made due to the unrealistic and speculative nature of the program.

	ROP gadgets	Vulnerable ROP gadgets	Proportion
Plain	66760	25297	37.9%
Exit points at IR-level	123251	87239	70.8%
Basic blocks	121632	84993	69.9%
	JOP gadgets	**Vulnerable JOP gadgets**	**Proportion**
Plain	112	60	53.6%
Exit points at IR-level	108	43	39.8%
Basic blocks	98	44	44.9%

Table 4.4: The number of gadgets for the simple program with many functions.

4.1.3 Redis

Performance analysis

Figures 4.1 and 4.2 display the performance of different Redis variants in response to a set of GET and SET requests. The performance stayed consistent for SET requests of up to 32 bytes and GET requests of up to 64 bytes in size. Bigger requests caused a steady drop in performance, which should not be a surprise – more data meant more network packets sent and a longer time needed to process them [2].

Redis exhibited the highest performance when it was not modified at all – nearly 1,600,000 requests per second for GET and 1,250,000 requests per second for SET, for 2-byte requests – since there was no overhead caused by additional instructions or supplementary memory accesses.

Adding instrumentation calls severely affected the performance; the recorded drop ranged between 34% and 44%, depending on the type and the size of the request. Performing a function call at every exit point was costly – the function arguments needed to be put onto the stack, which was followed by a control transfer. After the instrumentation function was executed, these steps had to be performed in reverse order. There were more optimal approaches than this.

Accessing memory at exit points at the source and the IR level was similar in terms of performance – as proven for the ‘Hello World’ program, in most cases, the same instructions were injected. Performing the memory access at the end of every basic block was slower, regardless of request type or size – a larger number of instructions was executed, and the memory was accessed more often, which lowered the overall performance.

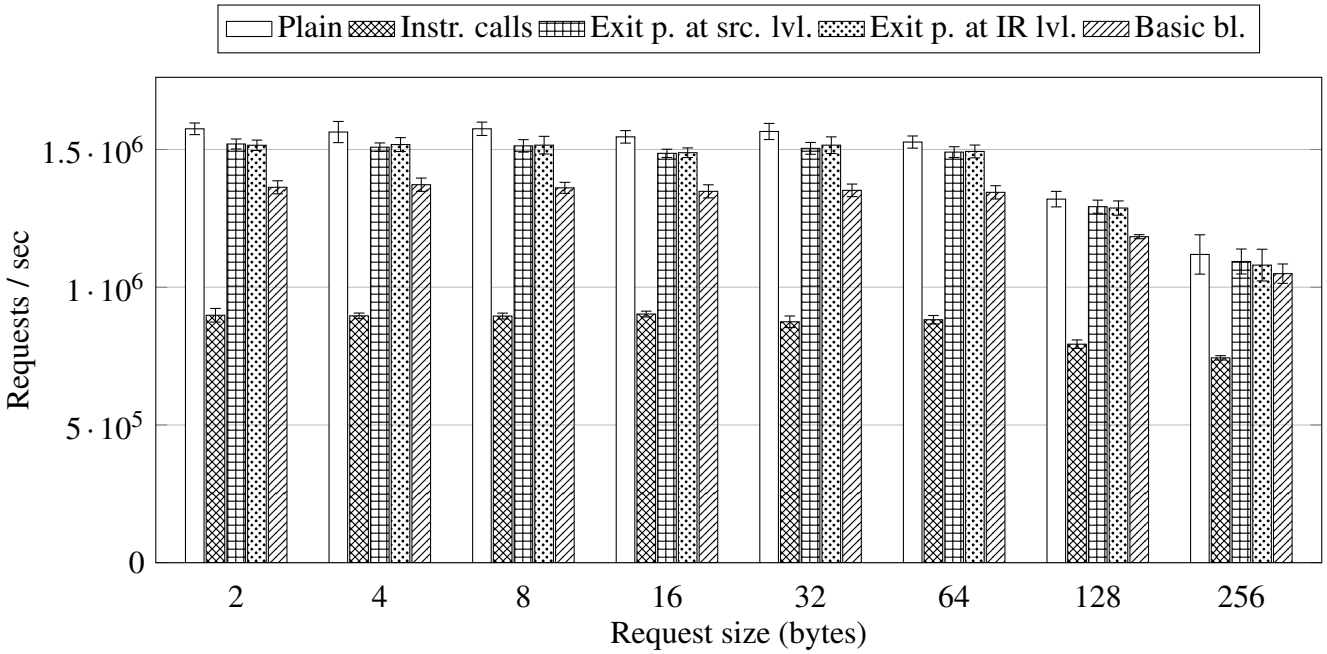

Figure 4.1: Redis performance in response to GET requests.

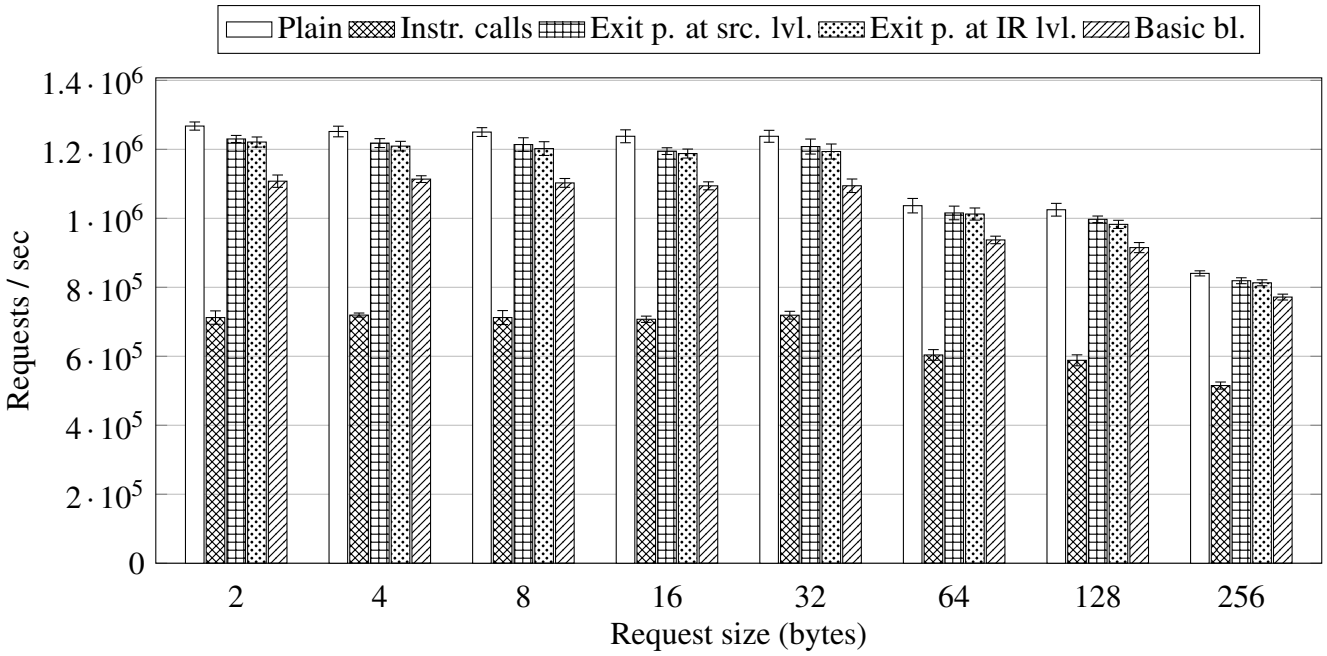

Figure 4.2: Redis performance in response to SET requests.

Assembly code analysis

A plain version of Redis consisted of over 385,000 x86-64 assembly instructions, excluding labels and assembly directives (see Table 4.5). Adding instrumentation functions increased its length by almost one-fifth; that was caused by the necessity to call them at every entry and exit point, which also entailed putting relevant arguments onto the stack and removing them later on. As evidenced in previous examples, this method caused the most considerable increase in code size, which could (but did not necessarily have to) have a detrimental impact on performance and code load time.

Protecting exit points at the source and the IR level yielded a change of, respectively, 4% and 3% in the total number of instructions. The impact of the modifications was similar, most likely due to their near-equivalence on the IR level, as explained in Section 4.1.1. However, adding minimal memory access on the source level generated a slightly higher number of instructions; that was caused by a minor inaccuracy in this method – it added memory access to inline functions. Once an inline function was embedded in the code, the memory access became redundant since it no longer protected any exit points.

As expected, protecting basic blocks was costly, with an increase of 13% in the number of instructions, but less costly than adding instrumentation functions.

	No. of instructions	**Difference**
Plain	385827	+0%
Instrumentation calls	470259	+22%
Exit points at source-level	401835	+4%
Exit points at IR-level	397899	+3%
Basic blocks	434372	+13%

Table 4.5: Static instruction count for Redis.

Gadget analysis

The number of ROP and JOP gadgets for Redis (Table 4.6) confirmed what was already shown for the ‘Hello World’ program – for its plain version, almost one-third of ROP gadgets and over a half of JOP gadgets did not access memory. That exposed over 45,000 ROP gadgets and over 3,500 JOP gadgets to a potential attacker.

Adding memory access at every exit point increased the overall number of ROP gadgets but also decreased the number of those that did not access memory – which was the intended behavior from

the beginning. This is, however, the only place where the initial expectations and the experiment results were consistent. Further observations:

1. Adding protections to all basic blocks resulted in fewer additional ROP gadgets than if the protections were applied only to exit points. One would expect that a higher number of supplementary instructions would have resulted in the opposite outcome.

2. Adding protections to exit points resulted in a decrease in the proportion of vulnerable JOP gadgets – from 27.1% to 24.4%. The initial assumptions stated that applying protections to exit points would protect against ROP attacks while applying them to all basic blocks would protect against both ROP and JOP attacks. The observed behaviour was contrary to those assumptions.

3. The change in the proportion of vulnerable JOP gadgets after applying protections to basic blocks was negligible – it stayed at about 24.4%. The protections were intended to guard against JOP attacks by decreasing that proportion, yet availed little.

The above observations were contrary to the initial assumptions of the experiment. The reason for that was simple: the impact of gadgets starting in the middle of existing instructions was vastly underestimated. The presence of such gadgets made the effects of applying protections much less predictable, and the protections themselves became inconsequential.

An attempt was made to create an ROP/JOP gadget chain using only gadgets that did not access memory. The chain was intended to simulate the `execve` function to create a shell, potentially granting the attacker a much wider range of motion. The attempt, however, failed – Ropper was unable to create such a chain.

	ROP gadgets	**Vulnerable ROP gadgets**	**Proportion**
Plain	72377	46057	63.6%
Exit points at IR-level	81545	42891	52.6%
Basic blocks	77544	41456	53.5%
	JOP gadgets	**Vulnerable JOP gadgets**	**Proportion**
Plain	13550	3671	27.1%
Exit points at IR-level	12764	3113	24.4%
Basic blocks	11611	2738	23.6%

Table 4.6: The number of gadgets for Redis.

4.2 Experiment summary

Initial findings

One of the proposed CFI enforcement methods, adding instrumentation calls, was quickly found to lack efficiency in terms of performance and the total number of assembly code lines. The need to perform extraneous function calls, half of which were redundant, was a significant factor leading to that conclusion, especially since, as it was soon found, similar behaviour could have been achieved at a much lesser cost. Additionally, it was discovered that protecting exit points led to similar changes to both assembly code and performance, regardless of whether it was done on the source or the IR level.

Performance analysis

Redis, one of the programs subject to the analysis, was tested in terms of performance before and after applying the proposed mitigation techniques. Protecting all exit points resulted in only a tiny drop in the program's efficiency; protecting all basic blocks imposed more severe penalties. Aside from the instrumentation calls, however, the impact of the proposed modifications did not render it unusable and could still allow them to be deployed in real-world applications.

Assembly code analysis

When analysing the number of assembly-code lines in the modified programs, it was found that adding memory access at every exit point most often resulted in only one instruction being added per exit point on both the source and the IR level. An analogous observation was made for protecting basic blocks – one additional instruction per basic block. While it may have initially seemed promising, it was also shown that for programs with many loops and conditional instructions, i.e., those with a high density of basic blocks, even these modifications could be detrimental.

Gadget analysis

Gadget analysis was meant to verify the experiment's primary assumptions: applying minimal memory access to all exit points would have decreased the number of vulnerable (i.e., those that do not access memory) ROP gadgets while applying it to all basic blocks would have done so to both ROP and JOP gadgets. These assumptions were instantly challenged upon revealing the research findings

– it was concluded that the program source code (or IR code, or assembly code, depending on the granularity one decides to assume) had little bearing on the number of gadgets. Instead of depending on code size or the number of basic blocks, their existence was purely dependent on the binary arrangement of the program since architectures with variable-sized instructions allow gadgets to start at any address. Hence, any attempts at estimating their number based on source code alone were very much missed.

For those same reasons, the attempts at applying the proposed protections did not result in any satisfactory outcomes – the number of vulnerable gadgets rarely decreased in response to the changes; in some cases, it even increased significantly. The reason for such unexpected gadget behaviour has already been explained; the introduction of mitigation techniques availed nothing more but providing additional code that could be exploited in ROP and JOP attacks. No amount of simple code injection, as done throughout the experiment, could have prevented the creation of gadgets on binary level. It might still be possible, but more sophisticated methods are required, and further research would be needed to explore them.

Regarding gadget chains, it was impossible to create one for Redis using only gadgets that do not access memory. It is generally assumed that conventional gadget manipulation tools are incapable of achieving that because of how uncommon of a restriction it is. However, Section 5.1 suggests that it might be possible regardless; additional study is necessary to establish the verity of that claim.

Faulty assumptions

Overall, the failure to anticipate the undesired gadget behaviour was caused by a missed assumption – it was believed that the number of gadgets composed of the fragments of 'intended' code dominated the total number of gadgets and that it was, therefore, safe to assume that it could be efficiently reduced by applying simple code injections. On the contrary, the problem at hand transcended the initial expectations and now necessitates further research in order to be tackled efficiently.

4.3 Limitations

1. Redis consisted of six libraries split across three compartments. Injecting IR code into some of them caused an unexpected system crash, most likely due to default FlexOS behaviour in response to a CFI violation. Despite several attempts at finding the problem's root cause, the specific location of the violation could not be identified. For that reason, two libraries remained

unmodified; the experiment was run regardless. Time constraints did not allow for further investigation to be conducted.

2. The agreed notion of 'minimal memory access' described in detail in Section 3.1 is not entirely correct. It has been mentioned that accessing a global variable would be sufficient to ensure that its owner (i.e., the relevant compartment) is the same as the entity attempting to access it. If it is not true, a CFI violation is detected, and the system crashes. However, FlexOS allows a single file to contain data belonging to multiple different compartments – that is achieved by proper use of built-in annotations and directives [33]. In that case, what compartment would a single global variable belong to? The most proper solution would be to declare one variable per compartment. Achieving it, however, would have required disproportionate amounts of effort for minimal gain; the impact on performance would have been minuscule. Therefore, the approach established in Section 3.1 was deemed acceptable.

3. Injecting IR code did not have the desired impact on the number of ROP and JOP gadgets. The reasons for that were twofold: first, since IR was yet to be compiled into assembly code (see Figure 2.1), it was not guaranteed that the injected code would stay at the desired place, i.e., before the `ret` instruction, or at the end of a basic block. Indeed, once the program was compiled, the modification could find itself in a location where it no longer protected against anything. Second, x86-64, as a CISC architecture, allows for variable-sized instructions [54]. That means that gadgets could start at any address and still be likely considered a valid sequence of instructions [45, 9]; no amount of IR code injections could protect against that. The only gadgets that could be purposefully suppressed with this kind of approach were the ones consisting of the 'intended' assembly code – injecting memory access in the middle of such a gadget would have indeed prevented it from being created. However, these situations were rare, as evidenced by the experiment results. It was as likely to prevent a gadget from being created by injecting additional code as it was to inadvertently allow for the creation of more gadgets.

Chapter 5

Possible exploits

So far, it has been indicated that there exists a vulnerability in FlexOS and that it could be exploited. The experiments run during the research process were meant to investigate whether Return- and Jump-Oriented Programming could be used as a foundation for a potential attack. The notion of gadget chains was briefly mentioned and quickly debunked as not viable, with the already existing protections in FlexOS. That is, however, one of many ways that ROP and JOP gadgets could be used to potentially exploit the vulnerability. The following sections outline rough ideas on how the exploits could be performed. While the feasibility of producing gadget chains has already been investigated, the other ideas are left for further research.

5.1 Gadget chains that do not access memory

Assume the potential attacker gains control of a compartment – they have access to all data contained therein. They can execute any function as long as it does not use libraries in other compartments. Any attempt to access data in non-compromised memory domains, however, will result in a system crash.

Now suppose the attacker has complete knowledge of the source code of other compartments. In that case, they can quickly identify ROP and JOP gadgets that do not access memory and are, therefore, exploitable regardless of the memory domain they are located in. Those gadgets can then be chained together to execute potentially malicious behaviour. The ROP version of this attack is visualised in Figure 5.1. Gadget chains are commonly used to simulate the `execve` function to spawn a shell [45]. With the restrictions already imposed by FlexOS, it might not be possible to create such a chain, for both ROP and JOP. Indeed, a tool commonly used for that purpose, Ropper, failed to

achieve it. There could, however, be a way to circumvent the issue of restricted memory access, at least for ROP attacks – using a memory stack instead.

When performing an ROP attack, the attacker has complete control of the stack. If they invoke a function in a non-compromised compartment, and that function accesses data on the stack, FlexOS does not flag the operation as unauthorised. That is also why 'minimal memory access' does not use local variables (for further details, refer to Section 3.1). If this structure is big enough, it could potentially be used in place of main memory, as long as gadget operations do not reference memory directly. Relevant ROP gadgets could then be as follows:

1. `mov %reg, %esp; ret;` – this gadget saves the stack pointer to the register `%reg`; it also makes it possible to define memory addresses relative to the top of the stack and reference data on the stack itself (e.g., by adding immediate values to `%reg`).

2. `mov [%reg1], %reg2; ret;` – this instruction sequence stores the contents of `%reg2` to the address pointed to by `%reg1`.

3. `mov %reg1, [%reg2]; ret;` – this gadget loads the data from the address pointed to by `%reg2` and stores it in `%reg1`.

However, this approach cannot be used in practice due to the commonly used protections against making memory regions both writable and executable [20]; the relatively small stack size compared to the main memory poses a problem as well. That being said, further research would be needed to indisputably discard this possibility.

5.2 Abusing gadgets to explore memory landscape

The above exploit description is optimistic by assuming that the attacker has complete knowledge of the contents of non-compromised compartments. In practice, it may not be the case; the issue of discovering those contents becomes a challenge by itself. This idea presents an attempt at tackling this issue.

When oblivious of the source code of other compartments, the attacker might take a completely blind approach and try to create ROP and JOP gadgets starting at a random address. Two things can then happen:

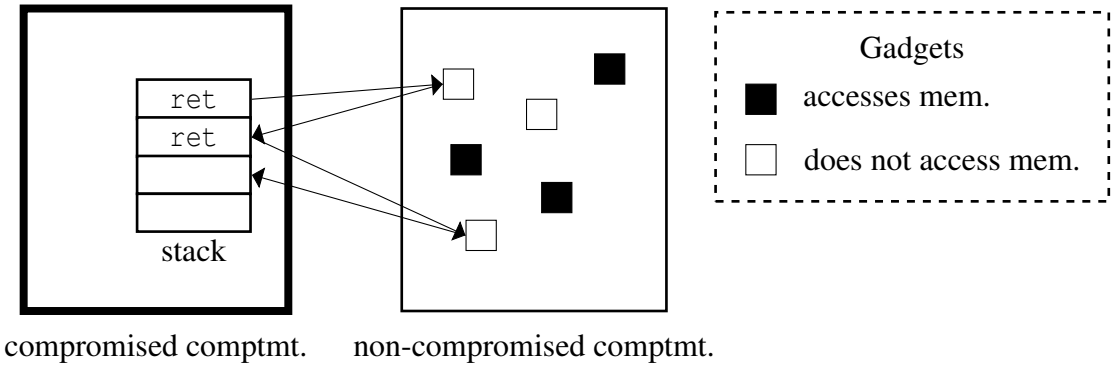

Figure 5.1: A visualisation of ROP gadget chaining. After gaining control of a compartment, the attacker places carefully chosen return instructions on the program stack. Removing such an instruction from the stack triggers the execution of a gadget being pointed to by ret. If the gadget does not access any memory, it can be executed regardless of the compartment it is located in.

1. Miss. The instructions starting at the selected address might (1) be invalid, (2) not form a valid gadget, or (3) form a valid gadget that accesses compartment-specific memory. In all three cases, further actions by the attacker would be prevented.

2. Hit. A valid gadget can be created at the selected address. While the attacker cannot see the specific instructions that form the gadget, they can deduce them by observing its behaviour. For example, `pop %reg; ret;` removes data from the top of the stack and puts it in `%reg`. The attacker, having complete control of the stack, notices that the data is gone, and while they do not know the specific register used, they are now aware of possible byte sequences starting at the selected address.

This method can yield minimal gain for excessive amounts of effort. Firstly, it is rare for a random address to mark the beginning of a gadget; more often than not, an attempt to perform the attack would result in a system crash. In order to be able to perform it repeatedly, the system must be able to reboot quickly and many times in a row. Secondly, even if a gadget is encountered, its behaviour might not be indicative of its specific structure, instead giving only a vague idea of the instructions comprising it.

A skilled attacker could potentially find a way to choose addresses in a more deterministic manner than ‘at random,’ and, upon finding a gadget, could devise a method to extract more information from its behaviour than described here. In order to achieve that, however, further research is required.

Chapter 6

Conclusion and future work

FlexOS is currently exhibiting a vulnerability that makes it possible to execute functions in any compartment as long as they do not access memory. Since that vulnerability can serve as a foundation for ROP- and JOP-based attacks, an effort has been made to mitigate it. Several methods of doing so were explored. Particular emphasis was put on their impact on performance and the number of assembly code lines to verify whether they are suitable for deployment in real-world programs. Additionally, a profound ROP and JOP gadget analysis was performed, investigating the impact of the proposed preventive measures on the overall number of gadgets and attempting to create potentially malicious gadget chains.

6.1 Future work

This report focused on mitigating a vulnerability in FlexOS with several approaches, none of which could be considered genuinely successful. The vulnerability persists, and so must the inquiry into mitigating it. In particular, several research directions are proposed.

6.1.1 More sophisticated methods of code injection

It was shown that performing memory access was not a viable way of minimising the number of vulnerable gadgets – applying the same instruction at all exit points and the ends of all basic blocks had an exactly opposite effect. It might be possible, however, to design byte sequences such that, when disguised as seemingly meaningless instructions and injected into relevant locations in the code, they could genuinely prevent gadgets from being created. It would require a profound investigation into

the instruction set of the relevant architecture, followed by an inquiry into the byte structure of the most common gadgets.

6.1.2 The feasibility of potential attacks

Chapter 5 outlines two possible attacks exploiting the vulnerability in FlexOS. While it is believed that simple gadget chaining, as done by conventional tools, might not be possible, an alternative approach was proposed. That, coupled with the idea of abusing gadgets to explore the memory landscape, shows that the true impact of the discovered vulnerability has not yet been thoroughly investigated. In order to design more efficient preventive measures, one would have to explore the feasibility of the already existing ideas and determine whether there are any other ways the vulnerability could be exploited.

6.1.3 Architectures with fixed-size instruction width

The biggest flaw of the experiment's assumptions was the belief that gadgets comprising 'intended' code were common and, therefore, easy to recognise. However, those assumptions failed when it was discovered that gadgets could start at any address in architectures with variable-sized instruction width. The proposed approach may well be correct as far as fixed-width instructions are concerned; certain research has already been done exploring this topic [29, 15]. In order to verify this claim, however, the experiment itself needs to be reworked – Memory Protection Keys are, after all, relevant to Intel processors, which by themselves largely support the x86 architecture.

6.2 Project outcomes

FlexOS is an important part of the research into system security. Because of its strong focus on easy switching between different back-end implementations, it stands out from other software enforcing security through isolation. By contributing to its development, scientists pioneer a largely unexplored area of research. This project, in particular, aided those efforts in several ways:

1. Investigated the impact of a potentially severe vulnerability in FlexOS.
2. Explored possible ways of exploiting said vulnerability.
3. Proposed techniques to mitigate the vulnerability and examined their effectiveness.

4. Laid solid groundwork for further research.

Bibliography

[1] *wc(1) Linux User's Manual*, June 2010.

[2] *Networks and Performance*, page 23–24. Performance and Tuning Series: Basics. Sybase, Inc., 2011.

[3] *llc(1) Linux User's Manual*, June 2017.

[4] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security*, 13(1):1–40, October 2009.

[5] ARC VeriTLA+. *Coccinelle, Version 1.1.1*. Nantes, France, 2021.

[6] Alvin Ashcraft, Matt Wojciakowski, Kent Sharkey, David Coulter, Drew Batchelor, Nicholas Adman, Erik Swan, Mike Jacobs, and Michael Satran. Control Flow Guard - win32 apps, Feb 2022.

[7] Markus Bauer and Christian Rossow. Cali: Compiler-assisted library isolation. ASIA CCS '21, page 550–564, New York, NY, USA, 2021. Association for Computing Machinery.

[8] Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. Wedge: Splitting applications into reduced-privilege compartments. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'08, page 309–322, USA, 2008. USENIX Association.

[9] Tyler Bletsch, Xuxian Jiang, Vince W Freeh, and Zhenkai Liang. Jump-oriented programming. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, New York, NY, USA, March 2011. ACM.

[10] Ajay Brahmakshatriya, Piyus Kedia, Derrick P. McKee, Deepak Garg, Akash Lal, Aseem Rastogi, Hamed Nemati, Anmol Panda, and Pratik Bhatu. Confllvm: A compiler for enforcing data confidentiality in low-level code. In *Proceedings of the Fourteenth EuroSys Conference 2019*, EuroSys '19, New York, NY, USA, 2019. Association for Computing Machinery.

[11] Michael D. Brown, Matthew Pruett, Robert Bigelow, Girish Mururu, and Santosh Pande. Not so fast: understanding and mitigating negative impacts of compiler optimizations on code reuse gadget sets. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1–30, October 2021.

[12] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *13th USENIX Security Symposium (USENIX Security 04)*, San Diego, CA, August 2004. USENIX Association.

[13] Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 385–399, San Diego, CA, August 2014. USENIX Association.

[14] Charly Castes. Diving into Intel MPK, Feb 2020.

[15] Tobias Cloosters, David Paaßen, Jianqiang Wang, Oussama Draissi, Patrick Jauernig, Emmanuel Stapf, Lucas Davi, and Ahmad-Reza Sadeghi. Riscyrop: Automated return-oriented programming attacks on risc-v and arm64. In *Proceedings of the 25th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '22, page 30–42, New York, NY, USA, 2022. Association for Computing Machinery.

[16] Jonathan Corbet. Memory protection keys, May 2015.

[17] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, page 555–566, New York, NY, USA, 2015. Association for Computing Machinery.

[18] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation. In *2014 51st ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, 2014.

[19] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the gadgets: On the ineffectiveness of Coarse-Grained Control-Flow integrity protection. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 401–416, San Diego, CA, August 2014. USENIX Association.

[20] de Raadt, Theo. amd64 kernel W$\oplus$X, Jan 2015.

[21] Artem Dinaburg. Let's talk about CFI: clang edition, Dec 2016.

[22] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. CCS '15, page 901–913, New York, NY, USA, 2015. Association for Computing Machinery.

[23] Jinyu Gu, Xinyue Wu, Wentai Li, Nian Liu, Zeyu Mi, Yubin Xia, and Haibo Chen. Harmonizing performance and isolation in microkernels with efficient intra-kernel isolation and communication. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*, pages 401–417. USENIX Association, July 2020.

[24] Khilan Gudka, Robert N.M. Watson, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Ilias Marinos, Peter G. Neumann, and Alex Richardson. Clean application compartmentalization with SOAAP. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, October 2015.

[25] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. Hodor: Intra-Process isolation for High-Throughput data plane libraries. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 489–504, Renton, WA, July 2019. USENIX Association.

[26] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, June 2013.

[27] IBM. Managing code size, Sep 2007.

[28] Intel. *Reducing Code Footprint in Embedded Systems*, page 185–186. Intel, 2022.

[29] Georges-Axel Jaloyan, Konstantinos Markantonakis, Raja Naeem Akram, David Robin, Keith Mayes, and David Naccache. Return-oriented programming on risc-v. ASIA CCS '20, page 471–480, New York, NY, USA, 2020. Association for Computing Machinery.

[30] Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh, and Dmitry Ponomarev. Branch regulation: Low-overhead protection from code reuse attacks. In *Proceedings of the 39th Annual International Symposium on Computer Architecture*, ISCA '12, page 94–105, USA, 2012. IEEE Computer Society.

[31] Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. Code-Pointer integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 147–163, Broomfield, CO, October 2014. USENIX Association.

[32] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[33] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Alexander Jung, Stefan Lucian Teodorescu, Sebastian Rauch, Felipe Huici, Costin Raiciu, and Pierre Olivier. FlexOS: towards flexible OS isolation. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, February 2022.

[34] Hugo Lefeuvre, Vlad-Andrei Bădoiu, Ştefan Teodorescu, Pierre Olivier, Tiberiu Mosnoi, Răzvan Deaconescu, Felipe Huici, and Costin Raiciu. Flexos: Making os isolation flexible. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '21, page 79–87, New York, NY, USA, 2021. Association for Computing Machinery.

[35] Jinku Li, Zhi Wang, Xuxian Jiang, Michael Grace, and Sina Bahram. Defeating return-oriented rootkits with 'return-less' kernels. In *Proceedings of the 5th European conference on Computer systems*. ACM, April 2010.

[36] Liu Liang. Data-execution prevention technology in windows system. *Information Security and Communications Privacy*, 2013.

[37] Joshua Lind, Christian Priebe, Divya Muthukumaran, Dan O'Keeffe, Pierre-Louis Aublin, Florian Kelbert, Tobias Reiher, David Goltzsche, David Eyers, Rüdiger Kapitza, Christof Fetzer,

and Peter Pietzuch. Glamdring: Automatic application partitioning for intel SGX. In *2017 USENIX Annual Technical Conference (USENIX ATC 17)*, pages 285–298, Santa Clara, CA, July 2017. USENIX Association.

[38] Shen Liu, Gang Tan, and Trent Jaeger. Ptrsplit: Supporting general pointers in automatic program partitioning. CCS '17, page 2359–2371, New York, NY, USA, 2017. Association for Computing Machinery.

[39] Ali Jose Mashtizadeh, Andrea Bittau, Dan Boneh, and David Mazières. Ccfi: Cryptographically enforced control flow integrity. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS '15, page 941–951, New York, NY, USA, 2015. Association for Computing Machinery.

[40] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, June 2013.

[41] Paul Muntean, Matthias Neumayer, Zhiqiang Lin, Gang Tan, Jens Grossklags, and Claudia Eckert. Analyzing control flow integrity with llvm-cfi. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC '19, page 584–597, New York, NY, USA, 2019. Association for Computing Machinery.

[42] National Security Agency. PhD thesis, Jul 2015.

[43] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys. *CoRR*, abs/1811.07276, 2018.

[44] Redis Ltd. Redis benchmark, 2018.

[45] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming. *ACM Trans. Inf. Syst. Secur.*, 15(1):1–34, March 2012.

[46] Adrian Sampson. Llvm for grad students, Aug 2015.

[47] Vasily A. Sartakov, Stefan Brenner, Sonia Ben Mokhtar, Sara Bouchenak, Gaël Thomas, and Rüdiger Kapitza. Eactors: Fast and flexible trusted computing using sgx. In *Proceedings of*

the 19th International Middleware Conference, Middleware '18, page 187–200, New York, NY, USA, 2018. Association for Computing Machinery.

[48] Vasily A. Sartakov, Lluís Vilanova, and Peter Pietzuch. Cubicleos: A library os with software componentisation for practical isolation. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '21, page 546–558, New York, NY, USA, 2021. Association for Computing Machinery.

[49] Sascha Schirra. *Ropper, Version 1.13.8*. Castrop-Rauxel, Germany, 2022.

[50] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient In-Process isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694. USENIX Association, August 2020.

[51] Felix Schuster, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in c++ applications. In *2015 IEEE Symposium on Security and Privacy*, pages 745–762, 2015.

[52] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, page 298–307, New York, NY, USA, 2004. Association for Computing Machinery.

[53] Vedvyas Shanbhogue, Deepak Gupta, and Ravi Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *Proceedings of the 8th International Workshop on Hardware and Architectural Support for Security and Privacy*. ACM, June 2019.

[54] John Stokes. Classic.ars: An introduction to 64-bit computing and x86-64, Sep 2008.

[55] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow integrity in GCC & LLVM. In *23rd USENIX Security Symposium (USENIX Security 14)*, pages 941–955, San Diego, CA, August 2014. USENIX Association.

[56] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, Santa Clara, CA, August 2019. USENIX Association.

[57] Victor van der Veen, Dennis Andriesse, Manolis Stamatogiannakis, Xi Chen, Herbert Bos, and Cristiano Giuffrdia. The dynamics of innocent flesh on the bone: Code reuse ten years later. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, page 1675–1689, New York, NY, USA, 2017. Association for Computing Machinery.

[58] Preeti Wadhwani and Smriti Loomba. *Global Cyber Security Market*. Nov 2022.