Forward-Edge and Backward-Edge Control-Flow Integrity Performance in the Linux Kernel

Christian Resell

Thesis submitted for the degree of
Master in Informatics: Programming and Networks
60 credits

Department of Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2020

Forward-Edge and Backward-Edge Control-Flow Integrity Performance in the Linux Kernel

Christian Resell

Forward-Edge and Backward-Edge Control-Flow Integrity Performance in the Linux Kernel

http://www.duo.uio.no/

Printed: Reprosentralen, University of Oslo

Abstract

This thesis will provide an overview of modern software defenses applied to the Linux kernel on the x86_64 and arm64 architectures. The main focus is the performance of forward-edge and backward-edge control-flow integrity. A requirement for control-flow integrity is to compile the Linux kernel with clang. Forward-edge control-flow integrity has an addition requirement of compiling the kernel with link-time optimization. Thus it is natural to also examine the effect of compiling the Linux kernel with clang, and what the performance impact of enabling link-time optimization is.

The results gathered from the benchmarks show that link-time optimization provides a performance boost in almost every single macro benchmark. The results also show that control-flow integrity comes at a cost. Kernels with control-flow integrity enabled perform worst in virtually all the benchmarks. Surprisingly, gcc performed better in the micro benchmarks, but performs slightly worse than clang on the macro benchmarks.

Acknowledgements

First of all I would like to thank my supervisor, Knut Omang, for letting me choose my own topic for this thesis. His patience and great advice have been crucial for completing the thesis.

Thanks to Aleksi Luukkonen for letting me use his hardware during the early stages of writing. This was very helpful for getting started. Thank you to Christoffer Buen for being a great rubber duck when I was stuck early in the research process.

I would also like to thank my family for all the support they have given me through the years. Finally, I would like to thank my girlfriend, Marit Iren Rognli Tokle, for supporting me and putting up with me while I have been working on this thesis.

June 15. 2020.
Christian Resell

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Introduction

Software security is a cat-and-mouse game between attackers and defenders. As more and more sophisticated protections are put in place, attackers come up with more creative and advanced techniques for circumventing them.

This thesis will look into modern software defenses employed today. First going through standard mitigations supported by most modern compilers, then finally looking at state-of-the-art protections applied to the Linux kernel. More specifically, our focus will be so-called forward-edge and backward-edge control-flow integrity implementations.

The modern defenses examined requires that the Linux kernel is compiled using clang, which is not the default compiler usually used to build the kernel. Thus it makes sense to examine whether switching to clang can provide a performance boost, or if it will slow the kernel down. To support control-flow integrity, clang uses something called link-time optimization, which is an interprocedural optimization technique that enables the compiler to apply optimizations across translation units. Since link-time optimization is a requirement for the chosen defenses to work, we will examine the impact it has on the kernel.

1.2 Background and motivation

The motivation for this thesis is to first examine modern software defenses and then looking at the performance impact of some of these. Whether or not a defensive measure is worth enabling depends on the performance impact and what sort of protection it provides. In some situations, it might be acceptable with a 10 % performance hit for increased security, while other times this is not acceptable at all. This thesis will not try to justify the choice of using a certain protection mechanism or not, but will provide an overview of the performance impact.

The reason for choosing the Linux kernel as the target system is because it is a complex open-source software project that runs on billions of devices. The recent development with support for clang and interesting software defenses make it a relevant choice for this thesis.

The security features we will discuss can be quite complex, and the same applies for the Linux kernel. We will spend some time introducing relevant background material in the earlier chapters so that the reader is more familiar with modern software security before delving into more complex topics.

1.3 Terms and definitions

This section will introduce some terms and definitions that do not fit in anywhere else, and are important for understanding the content of this thesis. First, we will look into the terms vulnerability and exploit in section 1.3.1. We will then clarify the terms kernel space and user space in section 1.3.2. In section 1.3.3 we will describe system calls. Finally we will discuss inlining in section 1.3.4.

1.3.1 Vulnerability and exploit

The terminologies *vulnerability* and *exploit* will be used quite a bit in later chapters when we discuss security mitigations. Thus, following is a short explanation of the difference between a vulnerability and an exploit. The IETF RFC4949 [70] defines a vulnerability as "A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy." An exploit is, for example, a piece of code that takes advantage of a vulnerability to compromise a system. Vulnerabilities will be discussed in chapter 3.

1.3.2 Kernel space and user space

Throughout this thesis we will discuss differences and similarities between *kernel space* and *user space*. Modern operating systems often have a separation between core components like drivers, process scheduling, and networking, and utilities like web browsers, text editors, and media players. User space refers to all software that runs outside of the operating system kernel, while kernel space refers to all code that runs inside the heart of the operating system.

1.3.3 System calls

A *system call*, or *syscall* for short, is a way for user space code to request services from the kernel. Examples include opening files, establishing a TCP connection, or creating processes.

1.3.4 Inlining

Inlining is an optimization compilers can perfom by replacing a function call site with the body of the function. This can eliminate some of the performance overhead associated with calling a function. However, too much inlining can have negative effects on performance because of increased code size. The programmer can also signal to the compiler that a function should be inlined by using the `inline` keyword. In the following code snippet the `foobar()` function is marked as `inline`.

```
inline int foobar(void)
{
        return 0x41414141;
}
```

The compiler does not have to inline the function, however, this is simply a hint to the compiler.

1.4 Conventions

For clarity, some conventions used throughout thesis are written down in this section. When refering to certain programming terms like variable types, assembly instructions, and so on, they are written using a `monospace font like this`. Function names are written using a monospace font and parentheses at the end like this: `function()`. File paths are also written using a monospace font: `/foo/bar/foobar.c`.

1.5 Outline

The rest of the thesis is structured as follows:

Chapter 2 will present some background information on the Linux kernel. First some brief history about Linux, Ubuntu, and Android, followed by an introduction to the two CPU architectures that are relevant for this thesis, namely x86_64 and arm64. Finally, Android IPC will be briefly discussed as this is the target for benchmarks performed on Android.

Chapter 3 will describe software vulnerabilities, defenses, and attacks. The chapters start with a general introduction and then moves on to memory corruption bugs and different techniques used by attackers. Then we will discuss some general memory corruption defenses and finally other vulnerability classes. A general understanding of these topics is required for the further chapters.

Chapter 4 will continue the general discuss about software vulnerabilities with more specific implementation details related to defensive mechanisms. The chapter will detail how certain defenses are implemented in the gcc and clang compilers. This will be useful for understanding the later chapters where we will delve into implementation details in the Linux kernel.

Chapter 5 describes some Linux kernel internals. First we describe the different versions of the kernel that are available, and then briefly introduce the build system. Finally we will discuss security features in the kernel.

Chapter 6 presents how forward-edge and backward-edge control-flow integrity is implemented in the Linux kernel. Since building the kernel with clang is a requirement for this to work at the moment, we will first discuss how this can be accomplished, and what work has been done to make it possible.

Chapter 7 describes the different performance benchmarks performed. We start off with presenting motivation and previous work before delving into the specific benchmarks chosen for this thesis.

Chapter 8 concludes the thesis with results from the benchmarks and presents ideas for future work.

Appendix features source code, scripts, and other information.

Chapter 2

Linux kernel

2.1 Introduction

In this chapter we will discuss Linux [45], and more specifically, the Linux kernel. In section 2.2 we will briefly look at the history of the Linux kernel. Then, in section 2.3, we will look at Android, which is Google's mobile operating system based on Linux. In section 2.4 we will briefly introduce Android's IPC mechanism, called Binder. In section 2.5 we will briefly discuss open source and how the Linux kernel is licensed. Finally, we will dive into details surrounding the different CPU architectures supported by Linux in section 2.6.

2.2 History and distributions

Linux is a free and open source operating system that works on everything from routers and tiny computers, to laptops, to large supercomputers. At the core of Linux is the kernel, which is the heart of the operating system. The kernel is responsible for scheduling processes, interfacing with hardware devices, and providing features for users.

Linux was created by Linus Torvalds in 1991, and since then has attracted many developers and companies to contribute to the project. According to the 2017 Linux Kernel Development Report [22] there were 1681 developers from 225 different companies contributing to the 4.13 version of the kernel. At that time, the kernel had 24,766,703 lines of code.

Linux is usually packaged in a so-called distribution, or distro for short, which is a fully-fledged operating system based on the Linux kernel. These distributions often come with their own package management system, where all sorts of software packages are distributed. Common components of a Linux distribution are the Linux kernel, GNU [30] tools and libraries, documentation, the X window system (or any other window system like Wayland), a window manager, and a desktop environment. There are hundreds of different Linux distributions in active development, so we will not look at all of these. The choice of distributions is based on their popularity, and that they vary a bit in their design, to showcase what type of Linux distros exist. First up is Ubuntu [17] which is a very popular user-friendly distro with commercial backing from Canonical Ltd. Ubuntu comes with a nice graphical user interface by default, and has support for proprietary drivers which makes it easier to use when dealing with hardware that has no or lacking support for open source drivers. Ubuntu is actually based on another distro called Debian [24]. While Ubuntu is a commercially backed distro, Debian is entirely driven by a community of volunteers. We could also have used Debian, since it is a popular distro that is the foundation for many other distros, like Ubuntu. In addition, it only consists of free software. Debian has a very long history as a distro, with version 0.01 released on September 15, 1993.

Debian Jessie and Stretch are two very stable distros that uses longterm support kernels. We will describe the different types of Linux kernel releases in section 5.2, and the different longterm kernels are listed in table 5.1. Debian Jessie relies on 3.16, which is supported until 2020, while Debian Stretch relies on 4.9, which is supported all the way until 2023. Ubuntu uses kernels that are a bit newer, 4.15 and 5.0 for versions 18.04 and 19.04, respectively.

We could have focused on other distros like Redhat, Fedora, Arch Linux, and Clear Linux as well, but Ubuntu is a very well tested and widespread distro that should cover all benchmarking needs. Some distros that rely on a rolling-release model might support more software that takes advantage of newer kernel features out-of-the-box, but the benchmarking software used in later chapters will not have any issues like this.

2.3 Android

The former section described different types of Linux operating systems primarily for servers and desktop computers. Linux is very widespread in the smartphone market because of Google's mobile operating system, Android [31], which is currently the most popular mobile operating system with 2 billion montly active users in May 2017 according to Google. Android introduces several modifications to the Linux kernel, and some of them eventually make it back to the upstream kernel. Binder, which is the *inter-process communication* (IPC) subsystem for Android, is an example of such a modification. It was originally developed by Be Inc., later Palm, Inc. for beOS among others. This code was later the based for Binder as we know it on Android today. Binder is a central component of Android, as the IPC mechanism is used by apps. We will describe Binder further in section 2.4.

Android updates are released on a yearly basis. All the major releases before 10 are named after different sweets, and as of June 15, 2020, Android 11.0 is the newest version, although it is currently released as a *preview* build. Table 2.1 lists all the different versions.

Many different companies like Samsung, Huawei, and Nokia use Android as the operating system for their smartphones. Google releases Android as the Android Open Source Project (AOSP), which other manufacturers can then modify to suit their needs. Many devices might need new drivers for device specific hardware, or other modifications that are not a part of AOSP.

Google develops their own smartphones, named Pixel. Currently, four generations of Pixel phones are released. The source code for Pixel and Pixel 3 is open source, and is part of AOSP. As we will discuss further in chapter 5, Google has introduced several interesting performance and security enhancements for their newer Pixel models.

Version	Name	Released
1.5	Cupcake	2009
1.6	Donut	2009
2.0	Eclair	2009
2.2	Froyo	2010
2.3	Gingerbread	2010
3.0	Honeycomb	2011
4.0	Ice Cream Sandwich	2011
4.1	Jelly Bean	2012
4.4	KitKat	2013
5.0	Lollipop	2014
6.0	Marshmallow	2015
7.0	Nougat	2016
8.0	Oreo	2017
9.0	Pie	2018
10.0	Android 10	2019
11.0	Android 11	2020

Table 2.1: Android versions

2.4 Android IPC — Binder

In this section we will briefly describe the binder IPC mechanism. Binder is commonly used to provide an IPC mechanism between apps. There are several binder domains (or contexts), however. The different domains were introduced to separate device-independent and device-specific code. Each domain has its own device driver and are isolated from each other. The domains available in current Android versions can be found in table 2.2 [6].

IPC Domain	Description
`/dev/binder`	IPC between framework/app processes
`/dev/hwbinder`	IPC between framework/vendor processes
	IPC between vendor processes
`/dev/vndbinder`	IPC between vendor/vendor processes

Table 2.2: binder domains

When we later discuss benchmarks on Android in section 7.1, binder and hwbinder will be used since there are benchmarking utilities available for these domains as part of the AOSP.

2.5 Open source

As stated earlier, the Linux kernel is an open-source project. This means that the code is freely available on the Internet for anyone to download and use. The Linux kernel is licensed under the GNU General Public License version 2 (GPLv2) [GPL]. The GPLv2 allows anyone to use the code, even in commercial products, as long as the source code is made available. Changes have to be tracked, and the modified source also has to be licensed under GPLv2.

The souce code for the Linux kernel can be found on `kernel.org` or on GitHub under Linus Torvald's account `torvalds` [1].

2.6 Architectures

Linux supports many different CPU architectures. The code for these are located in the `arch` directory in the top-level Linux kernel directory. As of 4.14, there are 31 architectures supported. For our purpose, however, we will only focus on x86_64 and arm64. x86_64 is the most common architecture for desktops and laptops, while arm64 are more common for smartphones and similar devices. x86_64 is also known as amd64, and arm64 is sometimes referred to as aarch64. These terms will be used interchangeably throughout the thesis. The 32-bit version of x86_64 is known as x86 and the 32-bit version of arm64 is known as arm. Although the research is not focused on x86 and arm, these architectures may be mentioned later, and x86 is used for some examples in later chapters. Since clang has very good cross-compilation support, porting this work to other architectures is definitely feasible. However, we will focus on x86_64 and arm64 here to save time and improve the quality of the research.

x86_64 and arm64 differ a lot in some of the low-level details of the architectures. For example how the instruction set is encoded, how the assembly code looks, the special CPU registers available, etc. The rest of this section will describe the differences we should be aware of when reading the rest of the chapters in this thesis.

One of the biggest differences is that arm64 is RISC-based, while x86_64 is CISC-based. The gist is that the arm64 *instruction set architecture* (ISA) contains fewer, and simpler instructions. In contrast with x86_64 which contains a lot of complex and specialized instructions. For RISC-based systems, each instruction is usually of the same length. CISC, on the other hand, employs a variable-length encoding.

Compared to x86_64, arm64 has a lot of registers. Table 2.3 summarizes the different registers available on x86_64 that are interesting for this thesis and their purpose. Note that the usage of certain registers may differ between operating systems, kernel space and user space, and so on. Table 2.4 summarizes the different registers on arm64 and their purpose. Note that these lists are not complete. They contain only the registers that are important for understanding the different code snippets, implementation details, etc. that follow in the next chapters. Some of these 64-bit registers also have a 32-bit version that allow you to access the lowest 32-bits of the register. For x86_64, these registers have the same name as they do on x86. Simply replace the `r` in their names with `e`. `rax` becomes `eax`, for example. For arm64, replace `x` with `w`. Thus, to access the lower 32-bits of `x0`, you would use `w0`.

2.6.1 Calling conventions

We will start by by examining how functions are called from assembly code. How arguments are passed to functions, and how the result is returned is known as

[1] `https://github.com/torvalds/linux`

name	purpose
rax	return value or system call number
rdi	first argument to function
rsi	second argument to function
rdx	third argument to function
rcx	fourth argument to function
r8	fifth argument to function
r9	sixth argument to function
rbp	base pointer
rsp	stack pointer
rip	instruction pointer

Table 2.3: x86_64 registers

name	purpose
x0	first argument to function or return value
x1	second argument to function
x2	third argument to function
x3	fourth argument to function
x4	fifth argument to function
x5	sixth argument to function
x8	system call number
sp	stack pointer
x29/fp	frame pointer
x30/lr	link register
pc	program counter / instruction pointer

Table 2.4: arm64 registers

```
static int sum(int a, int b)
{
        return a + b;
}

int main(void)
{
        int s = sum(13, 37);
        printf("sum: %d\n", s);

        return 0;
}
```

Figure 2.1: Sum code example

the *calling convention*. In this section, we will describe the calling convention for x86_64 and arm64 on Linux systems. Calling conventions may differ between different compilers and operating systems, but these descriptions are valid for Linux with the gcc and clang compilers. We will start this section by describing different calling conventions, and then move on to how they are implemented in low-level assembly code.

On x86_64, the first six function arguments are placed in `rdi`, `rsi`, `rdx`, `rcx`, `r8`, and `r9`. If the function needs more arguments, they are pushed on the stack. The return value is placed in `rax`. When issuing a system call, the arguments are placed in `rdi`, `rsi`, `rdx`, `r10`, `r8`, and `r9`. The system call number is placed in `rax`, and the return value also ends up there after the syscall has completed.

The calling convention for arm64 is easier to remember, as the first six arguments to a function are placed in `x0` to `x5`. The same registers are used for arguments to system calls. The return value from functions and system calls are placed in `x0`, and the system call number is placed in `x8`.

Consider the simple piece of code in figure 2.1 that calls a function named `sum` to add two numbers and then prints the result.

We now know that the function arguments will be placed in `rdi` and `rsi` for x86_64, and `x0` and `x1` for arm64. Let us take a look at how the assembly code looks like for these two architectures. The code can be found in figure 2.2, and it has been cleaned up a bit for clarity.

We will focus on x86_64 first. The first argument, 13, is placed is edi. `rdi` is not used here since `sum()` takes two `int` arguments. The size of an int on x86_64 is 32-bits, which means that it fits nicely in the lower part of `rdi`. The second argument, 37, is placed in esi. `call` is used to call into the function. Every `call` instruction (almost) is paired with a `ret`, the return instruction. When `call` is issued, the CPU actually performs several steps. First, the address of the next instruction, which is known as the return address, is pushed onto the stack. Next, the CPU jumps to the target of the call. In this case it is the `sum` function. Conversely, `ret` pops the return address off the stack and jumps there.

For arm64, the first argument is placed in `w0`, the second in `w1`. Here, the lower 32-bit part of `x0` and `x1` is used, just like for the x86_64 example. When calling a function, we now see a different assembly instruction, namely `bl`. `bl` is

```
main:
  mov  w0, #13
  mov  w1, #37
  bl   sum

sum:
  sub  sp, sp, #16
  str  w0, [sp, #12]
  str  w1, [sp, #8]


  ldr  w0, [sp, #12]
  ldr  w1, [sp, #8]
  add  w0, w0, w1

  add  sp, sp, #16
  ret
```

(a) arm64 sum example

```
main:
  mov  edi, 13
  mov  esi, 37
  call sum

sum:
  push rbp
  mov  rbp, rsp
  mov  dword ptr [rbp - 4], edi
  mov  dword ptr [rbp - 8], esi

  mov  esi, dword ptr [rbp - 4]
  add  esi, dword ptr [rbp - 8]
  mov  eax, esi

  pop  rbp
  ret
```

(b) x86_64 sum example

Figure 2.2: Calling convention comparison

```c
int do_stuff(int a, int b, int c)
{
        return sum(a, b) * c;
}
```

Figure 2.3: C code example of a non-leaf function

short for *branch with link*. This is one point where the two architectures differ a bit. On arm64, there is a special register called the link register. The `x30` register works as the link register, but it is sometimes simply referred to as LR. Instead of storing the return value on the stack, some arm64 functions will use `bl`, which instead stores the return value in `x30`. The `ret` function will consult this register to find the correct return address. Return addresses may also be stored on the stack, but in that case they are moved into `x30` before returning.

One thing to note here is that `sum` is a so-called leaf function. Leaf functions do not call any other functions, they are the leaves of the program's control flow graph. This means that there is no need to store the return address on the stack for this function, since it will not call any other functions, thus not touching `x30`. The `ret` instruction can then freely return without the need to fetch the return address from the stack. To demonstrate the difference between a leaf function and a non-leaf function, consider the code example in figure 2.3. We have added a new function to the program in figure 2.1 called `do_stuff()` that calls `sum()`, the leaf function. Calling leaf functions and non-leaf functions usually look the same, unless the compiler has performed some kind of optimization. But the prologues and epilogues will differ, which will be described in the next section. In addition to the `call` and `bl` instructions, we also have what is known as indirect calls. These calls go through a register or memory location.

```
do_stuff:
  # make room on the stack
  sub   sp, sp, #32
  # store the old frame pointer and link register
  stp   x29, x30, [sp, #16]
  add   x29, sp, #16

  # store first argument on the stack
  stur  w0, [x29, #-4]
  # store second argument on the stack
  str   w1, [sp, #8]
  # store third argument on the stack
  str   w2, [sp, #4]

  # load first argument to sum()
  ldur  w0, [x29, #-4]
  # load second argument to sum()
  ldr   w1, [sp, #8]
  # call sum()
  bl    sum
  # load third argument into w1
  ldr   w1, [sp, #4]
  # w0 = w0 * w1
  mul   w0, w0, w1

  # restore frame pointer and link register
  ldp   x29, x30, [sp, #16]
  # restore stack pointer
  add   sp, sp, #32
  # return
  ret
```

Figure 2.4: arm64 assembly code example of a non-leaf function

2.6.2 Prologues and epilogues

A prologue is the set of instructions executed at the start of a function. Conversely, the epilogue is the set of instructions that are executed at the end of a function. In this section we will refer to frame pointers, which point to *stack frames*. A stack frame corresponds to a function call, and contains the parameters, local variables, return address, and so on for the call. The stack frames for `do_stuff()` and `sum()` for the code example in figure 2.3 can be seen in figure 2.5. The stack grows upwards in the figure, i.e. towards lower addresses.

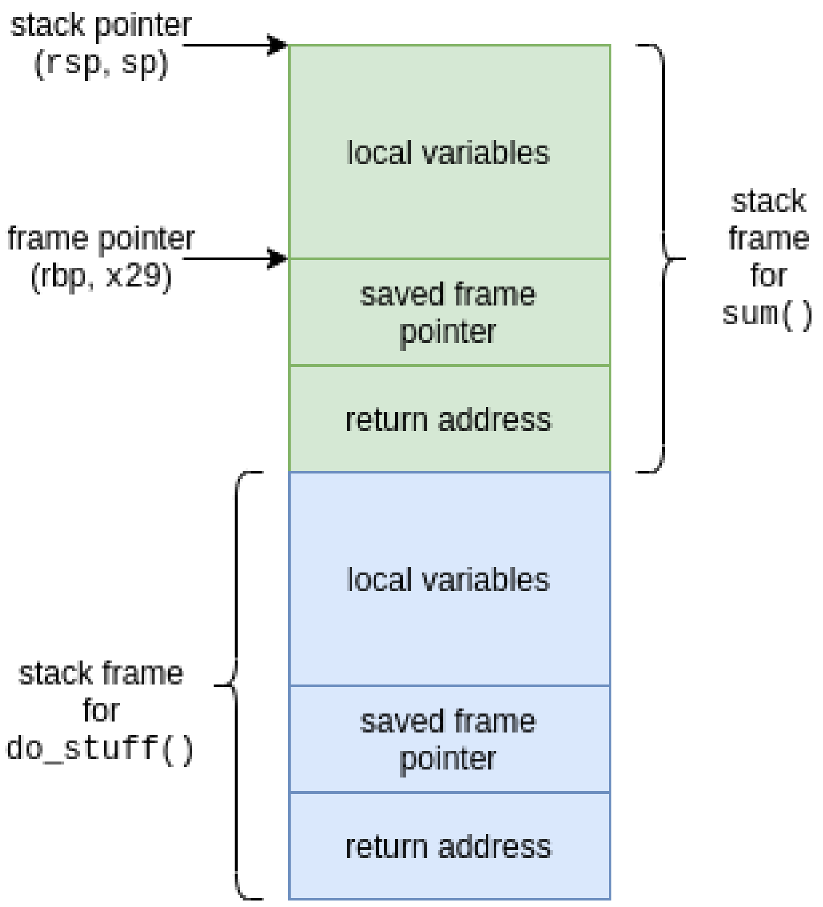


Figure 2.5: stack frame example

If we return back to the assembly code example in figure 2.4, we see that the first three instructions are there to prepare the stack. First, room is made on the stack to store local variables and the frame pointer and link register of the previous function. This function will make room for 32 bytes on the stack. The frame pointer and link register are stored at offset 16 and 24 from the new stack pointer, respectively. 16 is then added to `x29` to make it point to the saved `x29` value on the stack. The ARMv8 Programmer's Manual [7] states that "The frame pointer (X29) should point to the previous frame pointer saved on stack, with the saved LR (X30) stored after it. The final frame pointer in the chain should be set to 0.” The stack pointer has to be aligned on a 16-byte boundary, which explains why the compiler makes more room on the stack than is strictly necessary.

Chapter 3

Software vulnerabilities

3.1 Introduction

In this chapter we will discuss software vulnerabilities. More specifically, we will look at memory corruption bugs, which commonly lead to software vulnerabilities in software written in low-level languages like C and C++. Memory safety issues are still widespread today, even though the problem has been known for a long time. A well documented case of a buffer overflow vulnerability being exploited by an attacker was in 1988. The exploit was used by the Morris worm [64], which spread on the Internet, infecting approximately 6000 computers [28]. Even though these issues have been known for a long time, memory safety is still a difficult problem. The Chromium project reports that around 70 % of their serious security bugs are memory safety problems [66].

We start off by looking at the memory layout of a process on Linux in section 3.2. This section will introduce some important information about different memory regions and how executable files look like on Linux. Then we will move on to section 3.3, where we will describe different types of memory corruption bugs and techniques for exploiting them. To fully understand the vulnerability mitigation techniques described in coming chapters, it is important to understand why we need these mitigations, and what sort of benefits they can provide. In section 3.4 we will look into common exploitation techniques used by attackers. Then, in section 3.5 we will discuss some common memory corruption defenses. Finally, we will look at other vulnerability classes that are relevant in section 3.6. Having some familiarity with exploitation techniques and vulnerabilities will allow us to better understand the value of mitigations, and if they are worth applying.

3.2 Memory layout

Most of the concepts related to vulnerabilities and exploits are more or less universal for all platforms like Linux, Windows, and macOS, but we will focus on Linux since it is more relevant for this thesis. The executable file format on Linux is called *Executable and Linkable Format* (ELF) [53]. ELF files can be executable files, relocatable object files, core files, and shared libraries. An ELF file starts with a header and is followed by a program header table or a section header table, or both. The ELF header starts with the magic values `7f 45 4c 46`, where the hex values `45 4c 46` spell out `ELF`.

The program header table is an array of structures describing a segment or other information the system needs to load an ELF file. Examples of program header entries include loadable segments, like the executable code of the program and writable memory used to store global variables. Other entries include dynamic linking information, auxiliary information, and the state of the stack. The section header table is an array describing all the file's sections. Some section types include a symbol table which can be used for dynamic linking. Another important type of section is the string table, which is used to store strings used by the ELF file. For example names of functions imported from shared libraries or names of the different sections, like `.text` for the executable code, or `.rodata` for read-only data. Table 3.1 summarizes the different section types. Some of the section types, like the GOT and PLT, are further described in section 4.4.3.

section name	summary
.bss	Uninitialized data
.comment	Version control information
.ctors	Initialized pointers to the C++ constructor functions
.data	Initialized data
.dtors	Initialized pointers to the C++ destructor functions
.dynamic	Dynamic linking information
.dynstr	Strings needed for dynamic linking, like function names
.dynsym	Dynamic linking symbol table
.fini	Termination code (destructors)
.gnu.version	Version table
.gnu.version_d	Version symbol definitions
.gnu.version_r	Version symbol needed elements
.got	Global Offset Table (GOT)
.hash	Symbol hash table
.init	Initialization code (constructors)
.interp	Name of a program interpreter (usually ld-linux)
.line	Line number information for debugging
.note	Note Section format
.note.GNU-stack	Used to declare stack attributes
.plt	Procedure Linkage Table
.relNAME	Relocation information applying to section `NAME`
.relaNAME	Relocation information applying to section `NAME`
.rodata	Read-only data
.rodata1	Read-only data
.shstrtab	Section names
.strtab	Strings
.symtab	Symbol table
.text	Executable instructions

Table 3.1: ELF sections

For the rest of the thesis it helps to have a general idea of how an ELF is mapped to memory when a program is running. Most ELF binaries rely on a dynamic linker like ld-linux [54] to run. The job of a dynamic linker is to map the ELF into memory, load any libraries it depends on, perform other preparations, and finally run the program. The exception is for statically linked executables, where the binary does not depend on any external libraries. The `.interp` section contains the full pathname of the dynamic linker to use for the program. For statically linked binaries, there is no `.interp` section.

Linux provides functionality to view memory mappings of running processes through the `/proc` filesystem. For each process on a Linux system, a separate directory exists in `/proc` named after the process ID (PID) of that process. There is also a special symbolic link at `/proc/self` that always redirects to the current process opening that link. The file `/proc/<PID>/maps` allows us to view the memory mapping of a process. Let us look at how the memory mapping for the `cat` program looks like:

```
[1] 00400000-0040c000 r-xp 00000000 103:06 3145872              /bin/cat
[2] 0060b000-0060c000 r--p 0000b000 103:06 3145872              /bin/cat
[3] 0060c000-0060d000 rw-p 0000c000 103:06 3145872              /bin/cat
[4] 01983000-019a4000 rw-p 00000000 00:00 0                     [heap]
[5] 7fe73f4e0000-7fe73fab8000 r--p 00000000 103:06 14811734     /usr/lib/locale/locale-archive
[6] 7fe73fab8000-7fe73fc78000 r-xp 00000000 103:06 8138465      /lib/x86_64-linux-gnu/libc-2.23.so
[7] 7fe73fc78000-7fe73fe78000 ---p 001c0000 103:06 8138465      /lib/x86_64-linux-gnu/libc-2.23.so
[8] 7fe73fe78000-7fe73fe7c000 r--p 001c0000 103:06 8138465      /lib/x86_64-linux-gnu/libc-2.23.so
[9] 7fe73fe7c000-7fe73fe7e000 rw-p 001c4000 103:06 8138465      /lib/x86_64-linux-gnu/libc-2.23.so
[10] 7fe73fe7e000-7fe73fe82000 rw-p 00000000 00:00 0
[11] 7fe73fe82000-7fe73fea8000 r-xp 00000000 103:06 8126731     /lib/x86_64-linux-gnu/ld-2.23.so
[12] 7fe740076000-7fe740079000 rw-p 00000000 00:00 0
[13] 7fe740085000-7fe7400a7000 rw-p 00000000 00:00 0
[14] 7fe7400a7000-7fe7400a8000 r--p 00025000 103:06 8126731     /lib/x86_64-linux-gnu/ld-2.23.so
[15] 7fe7400a8000-7fe7400a9000 rw-p 00026000 103:06 8126731     /lib/x86_64-linux-gnu/ld-2.23.so
[16] 7fe7400a9000-7fe7400aa000 rw-p 00000000 00:00 0
[17] 7fffc1ca9000-7fffc1cca000 rw-p 00000000 00:00 0            [stack]
[18] 7fffc1d08000-7fffc1d0b000 r--p 00000000 00:00 0            [vvar]
[19] 7fffc1d0b000-7fffc1d0d000 r-xp 00000000 00:00 0            [vdso]
[20] ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0    [vsyscall]
```

The numbers in front have been added to make it easier to describe the different memory regions. 1-3 comprises the cat binary. 1 starts with the ELF header, but most importantly includes the `.text` section of the binary, which contains the executable instructions. Note that this region is marked as readable (`r`) and executable (`x`). 2 is read-only data, and 3 is data that can be read and written, for example the `.data` and `.bss` sections. Read-only data includes strings and other data that never changes throughout the lifetime of the program. At 4 we have the heap, which is used for dynamic memory allocations, for example memory returned from the `malloc()` function call.

The next section, 5, is not very important for our purposes, but for completeness we describe it here as well. The locale archive contains locales, which are "collections of language and country specific conventions allowing to adapt software to the user's preferences." [29]. It is loaded and used by the C library.

Sections 6-10 are related to the standard C library, or libc for short. 6 contains the executable code, and 7 is a section with no permissions. This might be present when loading certain libraries or binaries. Usually the whole ELF is loaded into memory with the memory protections being applied later by parsing the program headers of the ELF. If certain sections are not meant to be mapped, or if there is a gap between them, they end up being inaccessible. Other times these sections may show up when a program deliberately creates mappings with no permissions to protect adjacent data. These allocations are often known as *guard pages*. Allocating a guard page before a region containing important data can thwart some buffer overflows since the program will crash as soon as it tries to read from or write to the guard page. Section 8 contains read-only data, and 9 contains writable data. 10 is an anonymous memory mapping allocated using `mmap()`. An anonymous mapping is not backed by a file.

11-16 pertain to the dynamic linker, and it is very similar to the binary and libc, since this is also an ELF file. The next few sections, however, are different from the previous ones. At 17 we have the stack, which is used to store local variables, and other important information like return addresses.

The next three sections, 18-20, are related to virtual syscalls [16]. Simply put, virtual syscalls are faster syscalls implemented in user-space for performance reasons. The syscalls implemented here are read-only syscalls, meaning that they do not change any structures in kernel space, they simply read data. The `vdso` region contains an ELF file that the kernel maps into processes, and if you dump this memory region to disk you can parse it like any normal ELF file. Virtual syscalls include `gettimeofday()`, `getcpu()`, `time()`, and `clock_gettime()`. These were introduced as virtual syscalls since the overhead of context switching between user space and kernel space was too high for processes that frequently checked the current time.

3.3 Memory corruption bugs

Now that we have a little useful background information for understanding memory corruption bugs, we will dive into common vulnerabilities and ways of exploiting them. Memory corruption occurs when memory is changed in an unintended way, causing a program's behavior to deviate from normal execution. For an attacker, the goal is to change program behavior to carry out unintended actions. The holy grail for at attacker is to gain *arbitrary code execution*, meaning that full control over the original program has been achieved, and it is now possible to execute *any* code. This is often accomplished by hijacking the original control flow of the program to run other code than intended.

An example of a very common type of attack for a memory corruption bug is a *buffer overflow*. This is any attack that results from a piece of code that does not check the bounds for a buffer. Buffer overflows have been around for a long time. Exploitation of buffer overflows used to be focused on buffers located on the stack, which is what we will see in this section. Overflowing a buffer on the stack and corrupting important data is also known as *smashing the stack* [3]. The code snippet in figure 3.1 is vulnerable to a buffer overflow.

The `gets()` function does not check the size of `buf`, resulting in a stack buffer overflow (or stack overflow for short) when the user inputs more than 32 bytes.

What will happen if we write past the 32 bytes assigned to the buffer? The answer is dependant on the architecture the program is running on, but to simplify the discussion we will focus on `x86_64`. Local variables, like the buffer in the example above, are stored on the stack. In addition to local variables, other interesting values are also stored on the stack which we will discuss later in this section. Consider the following assembly code snippet:

```c
void function(void)
{
        char buf[32];

        printf("Give me some data: ");
        gets(buf);

        printf("data: %s\n", buf);
}
```

Figure 3.1: Buffer overflow source code example

```
        mov     rdi, buf
        call    gets
```

The first line moves `buf` into `rdi`, which is the first and only argument to the `gets()` function. On the next line, a `call` instruction is issued. When `gets()` is done, we want our program to continue executing the code after the `call` instruction. To accomplish this, the CPU automatically pushes the address of the instruction after the `call` to the top of the stack. This means that the stack contains return addresses. Other important data on the stack includes saved base pointers, stack cookies (see section 3.5.2), local variables, and so on. In short, there are many valuable targets on the stack for an attacker.

Before modern vulnerability mitigations came into play, it was possible to execute code on the stack. A common attack scenario was to overflow a vulnerable buffer on the stack, overwrite the saved return address to point into the same stack buffer and finally execute code injected into the stack. Today, a protection known as non-executable stack (NX) [79] prevents this from happening. More generally, this mitigation makes sure that no regions, like the stack and heap, are writable and executable at the same time by default. Such regions of memory still exist, particularly in programs that make heavy use of just-in-time (JIT) [9] compilation like modern web browsers. NX will be discussed further in section 4.4.4.

Shellcode

Code injected like this is often known as *shellcode*. Usually the goal is to spawn a shell, which is where the name comes from. However, shellcode is simply a sequence of instructions used to carry out any action desired, not just spawning a shell. When using shellcode in an exploit, depending on the constraints of the target program, several tricks may be necessary to make the shellcode work in the target environment. We can imagine attempting to exploit a stack overflow where the program uses `strcpy()` to copy data into a vulnerable buffer. `strcpy()` expects the data to be a valid C string, which means that it will stop reading data once a null byte is encountered. To work around this, the shellcode has to be *null free*, i.e. no null bytes can occur anywhere in the shellcode. For certain input functions, like the `scanf()` family of functions, reading stops when encountering any whitespace character. For example tabs, vertical tabs, or spaces. There might also be size constraints or other illegal bytes that cannot occur in the shellcode.

The shellcode presented later in this section corresponds to the following C snippet:

```
char **args = { "/bin//sh", NULL};
execve("/bin//sh", args, NULL);
```

`execve()` is the system call used to start new processes on Linux, which means that this program will spawn a shell. Note that the path `/bin//sh` is used instead of `/bin/sh`. The extra forward slash is there to make the shellcode free of null bytes. Following is the output of `objdump` for the shellcode, showing both the bytes used to encode the instructions, and the human readable assembly code.

```
$ objdump -D -b binary -m i386:x86-64:intel sc.bin

sc.bin:     file format binary


Disassembly of section .data:

0000000000000000 <.data>:
0:      48 31 c0                    xor    rax,rax
3:      50                          push   rax
4:      48 b9 2f 62 69 6e 2f        movabs rcx,0x68732f2f6e69622f
b:      2f 73 68
e:      51                          push   rcx
f:      48 89 e7                    mov    rdi,rsp
12:     50                          push   rax
13:     57                          push   rdi
14:     48 89 e6                    mov    rsi,rsp
17:     48 31 d2                    xor    rdx,rdx
1a:     6a 3b                       push   0x3b
1c:     58                          pop    rax
1d:     0f 05                       syscall
```

As with most shellcode, it can be hard to understand what is going on. This code starts by setting `rax` to zero by XORing with itself, and pushing it on the stack. Next, a weird-looking number is pushed onto the stack. This is the string `/bin//sh` encoded as a hexadecimal number. To make sure the string is null-terminated, the zero was pushed onto the stack earlier. Recall that syscalls on x86_64 require that the syscall number is stored in `rax` and the first three arguments are stored in `rdi`, `rsi`, and `rdx`. At address `f`, the first argument is set to point to the `/bin//sh` string, which means the first argument is now set up correctly. The next argument to `execve` is a little bit trickier since we have an array of pointers to C strings. The array has to end with a `NULL` pointer. First, `rax`, which is still zero, is pushed onto the stack again. `rdi` is then pushed, which points to `/bin//sh`. We now have two pointers stored next to each other on the stack, the first one pointing to our string, the second a `NULL` pointer. We can then set `rsi` to point to the stack. `rdx` is then set to zero, as we do not care about this argument. Finally, `rax` is set to `0x3b`, which is the syscall number for `execve`, and the syscall is executed, starting a shell.

When faced with NX, attackers had to come up with new and inventive solutions for bypassing it. The following section will describe code-reuse attacks, which can be used to defeat the NX protection.

3.4 Code-reuse attacks

As the name implies, *code-reuse attacks* will reuse code that already exists within a program to hijack the control flow. We will start by describing return-to-libc in section 3.4.1 and then move on to return-oriented programming in section 3.4.2. In section 3.4.3 we will take a look at another code-reuse attack known as jump-oriented programming. Finally, we will discuss other relevant attacks in section 3.4.4.

3.4.1 return-to-libc

Return-to-libc [25] [80] [77], also known as ret2libc, is a code-reuse attack technique. In this technique, an attacker will use a memory corruption vulnerability to return into functions in libc (or other loaded libraries). This allows an attacker to reuse existing functionality in those libraries to carry out an attack. In their paper [77] Tran et. al even show that return-to-libc attacks are Turing complete. A lot of interesting functionality is available in libc which can be leveraged by an attacker to get full control over an application under attack. One common target for attackers is the `system()` function which will execute any shell command passed to it. On x86 (32-bit), performing a return-to-libc attack is a little simpler than on x86_64, so we will use that as an example. Consider the previous example in figure 3.1, where the dangerous `gets()` function is used to read data from the user. The stack looks pretty similar on x86 and x86_64, but the big difference is that arguments are passed on the stack by default on x86. There are, however, some calling conventions where a limited amount of function arguments are placed in registers on x86, but this is not the case when calling functions from libc. Now consider the following C/asm source example for x86:

```
int main(void)
{
        system("/bin/sh");
        return 0;
}
```

(a) C: system("/bin/sh") example

```
main:
    push ebp
    mov  esp, ebp

    push  0xbadc0de ; "/bin/sh"
    call  system
out:
    xor   rax, rax
    leave
    ret
```

(b) x86 assembly: system("/bin/sh") example

Figure 3.2: Calling `system()`

The `out` label has been added to the assembly code for clarity. In this example, we assume that the string "/bin/sh" is located at address 0xbadc0de. Right before executing the call instruction, the stack layout can be seen in figure 3.3 [1]

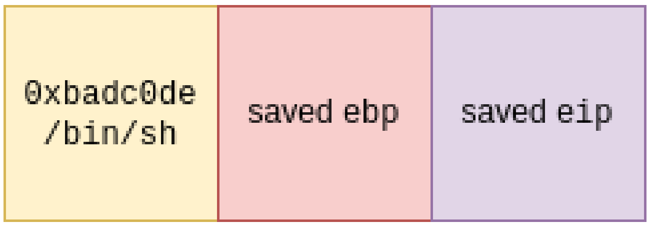


Figure 3.3: x86 stack frame before call

After the call instruction, the address of the next instruction, which is the xor instruction after the out label, will be pushed on the stack. The layout can be seen in figure 3.4.

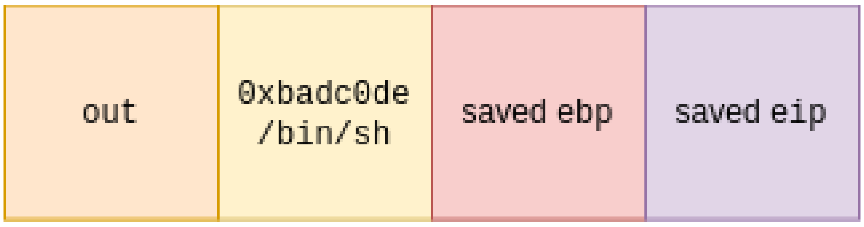


Figure 3.4: x86 stack frame after call

Now that we have an understanding of the stack layout when calling functions on x86, we can revisit the example in figure 3.1. We can imagine that the stack layout looks something like figure 3.5. When filling buf with 32 bytes

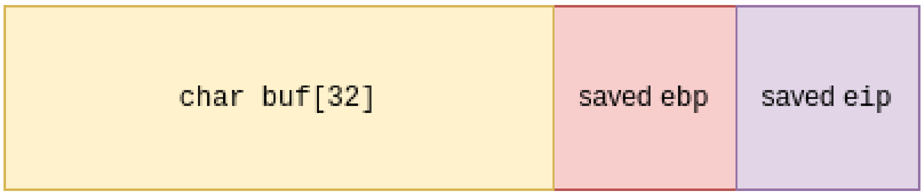


Figure 3.5: x86 stack frame for buffer overflow example

of data, any more written to the buffer will overwrite the saved base pointer (ebp) and then the saved return address (eip). If the saved return address is overwritten, control-flow will be hijacked when the function returns. From the system() stack layout example in figure 3.4 we know how the stack should look like when performing a function call. With the correct input, we can cause the program to call system("/bin/sh") instead of returning back to the original caller. See figure 3.6 for an example of how it may look like.

As mentioned in section 2.6.1, the first argument to functions in x86_64 is stored in rdi. On arm64, x0 is used. This means that a simple return-to-

[1]the arguments to main(): argc, argv, and envp have been omitted for simplicity.

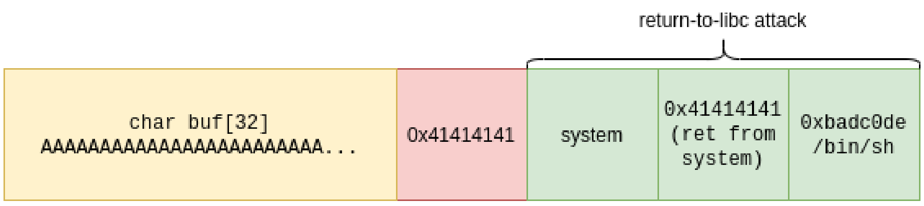


Figure 3.6: x86 return-to-libc stack frame

libc attack as shown here will not work on these architectures since parameters are not passed on the stack. To work around this challenge, return-oriented programming can be used. This technique is described in the next section.

3.4.2 Return-oriented programming

In *return-oriented programming* (ROP) [69] small sequences of code known as gadgets are used to carry out the wanted effect, bit by bit. As van der Veen et. al show in their paper [78], ROP is still a useful technique ten years after it was first introduced.

We will start by introducing ROP on x86_64, and then look at differences on arm64 at the end of this section. Gadgets usually end with a `ret` instruction, and are placed after each other on the stack. The first gadget will overwrite the saved return address. When the first gadget is done executing, the `ret` instruction will run, effectively fetching the next gadget from the stack and returning there. Following is an example of a gadget that sets the `rax` register to zero.

```
xor  rax, rax
ret
```

For the attack in figure 3.6 in the previous section to work, we would have to place a pointer to `/bin/sh` in `rdi/x0` before executing `system()`. To accomplish this, we could use a gadget like this:

```
pop rdi
ret
```

The `pop rdi` instruction will fetch the next value on the stack and place it into `rdi`. The following figure shows how the previous attack could look like when using ROP to control the argument in `rdi`. The gadget placement is highlighted in blue.

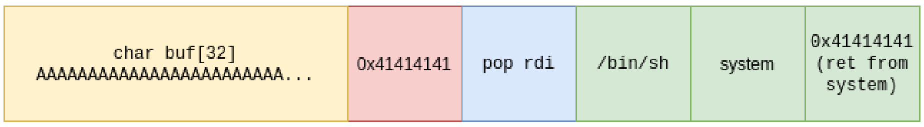


Figure 3.7: x86_64 ROP example

Within the target program and the set of loaded libraries, a lot of gadgets exist, making it very likely that an attacker will succeed in running arbitrary code as long as the memory address of the code containing the gadgets is known. As we will see in section 3.5.1, however, knowing the memory layout of a program might not be straightforward.

3.4.3 Jump-oriented programming

Jump-oriented programming [15] (JOP) is a class of code-reuse attack that does not rely on the stack and `ret` instructions. Instead of using gadgets ending with a `ret` instruction, JOP relies on gadgets ending with an indirect `jmp`. To maintain control between JOP gadgets, a *dispatcher gadget* is used. The job of the dispatcher gadget is to maintain a dispatch table containing the gadgets to be executed, and fetching the next gadget after the previous one is done. The following is an example dispatcher gadget from [15]:

```
add ebp, edi
jmp [ebp-0x39]
```

This dispatcher gadget uses `ebp` as its dispatcher table. First, `edi` is added to `ebp`. The code then jumps to the address stored at `ebp - 0x39`. If an attacker has control over the memory pointed to by `ebp` before executing this gadget, and `edi` is a reasonable value, it is possible to use this dispatcher gadget to chain together many gadgets. The gadgets will be stored in `ebp` with `edi` as the distance between them. If `edi` was 8, a dispatch table could look something like this:

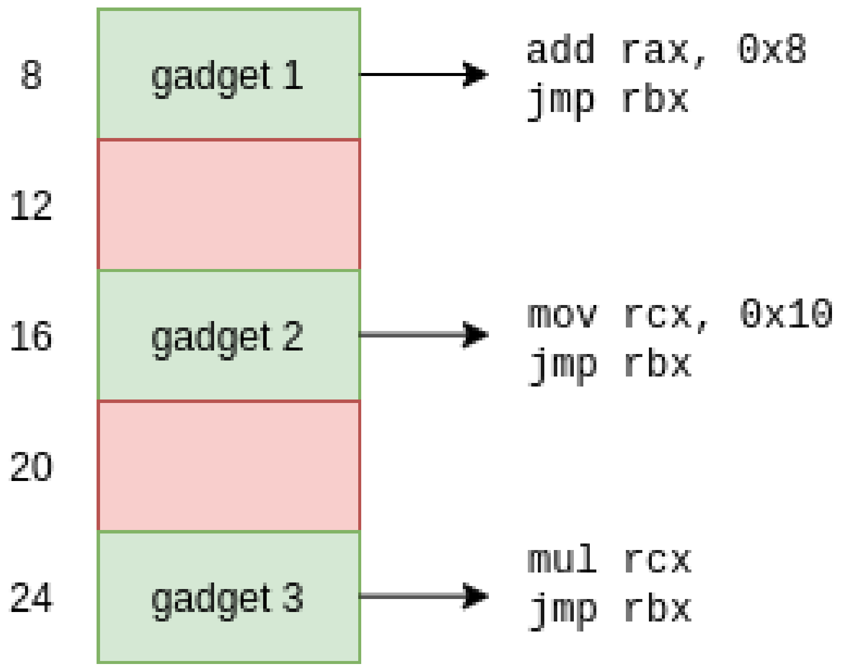


Figure 3.8: JOP dispatch table example

Consider a situation where `ebp-0x39` corresponds to the dispatch table above. `edi` is 8, and `rax` is 0. First, control-flow is hijacked by overwriting a function pointer stored in `rbx` to return to the dispatch gadget listed above. 8 is then added to `ebp` and the gadget jumps to gadget 1. Gadget 1 adds 8 to `rax`, setting it to 8. The gadget returns by jumping back to the dispatcher gadget. Next, 8 is added to the dispatch table again, effectively skipping the entry at offset 12 and fetching the next gadget at offset 16. The dispatcher then jumps to gadget 2 which sets `rcx` to 0x10 and then jumps back. Again, 8 is added to the dispatch table to fetch the final gadget, gadget 3. This gadget multiplies `rax` and `rcx`, storing the result in `rax`.

3.4.4 Other attacks

In addition to the previously mentioned attack techniques like ROP and JOP, there are many other similar techniques that exist. We will not go into these in detail, but some of them will be mentioned here briefly for completeness. *Data-oriented programming* (DOP) is a "general method to build Turing-complete non-control data attacks against vulnerable programs" [34]. DOP does not target data that is directly used in control transfer instructions, like function pointers used in indirect calls. Instead, other data is tampered with to make existing data flows in the program carry out unintended actions which can lead to a full compromise of the target program.

Counterfeit object-oriented programming (COOP) [68] is a code reuse attack that targets C++ programs. COOP relies on conterfeit objects, which are C++ class objects that are created by the attacker, and not by the original program. Since COOP targets programs written in C++ it is not very useful for this thesis since the Linux kernel is written in C.

3.5 Memory corruption defenses

The following sections will describe some important memory corruption defenses in more detail. In section 3.5.1 we will describe address space layout randomization, and in section 3.5.2 we will go into detail on stack protection.

3.5.1 Address space layout randomization

Modern operating systems have a security mechanism known as *address space layout randomization* (ASLR) [72] [12]. As the name implies, ASLR relies on randomization to make it harder for an attacker to know how the address space layout for a program looks. It works by randomizing the base address of the sections loaded into memory for executables. Regions that are randomized include the stack, the heap, and shared libraries. Other regions are also randomized, but they are not relevant for this thesis. Although ASLR is on by default for shared libraries, this is not always the case for binaries. For ASLR to work with binaries, it has to be compiled as a *position independent executable* (PIE). Libraries are position independent by default, which means that they do not rely on hard-coded addresses for accessing variables inside the program, and so on. ASLR is also enabled for the Linux kernel, where it is known as kernel ASLR (KASLR). This works by loading the kernel at a random address during boot. Because the kernel requires 16MB alignment, there are only around 512 different base addresses for KASLR which makes it weaker than ASLR in user space. However, if an attacker takes a wrong guess the kernel will probably crash, making it infeasible for an attacker to actually exploit a vulnerability where they have a 1/512 chance of success.

3.5.2 Stack protection

To prevent attackers from exploiting stack overflows, stack canaries [23], or stack cookies, were introduced. A stack canary is a random value placed on the stack to act as a barrier between local variables and the saved return address. The size of the stack canary matches the size of the CPU, e.g. 64-bit for x86_64 and arm64, and 32-bit for x86 and arm. The stack canary is placed on the stack during the function prologue, and then checked during the epilogue. If the stack canary does not match the expected value during the epilogue, the program will crash. This increases the difficulty for attackers in the presence of stack overflow vulnerabilities. Now, attackers would need an additional vulnerability that allows them to leak stack contents outside of the local variables, or they would have to somehow overwrite only the saved return address without corrupting the stack canary. To thwart some types of information disclosure bugs, the stack canary will include a null byte, to deter an attacker from leaking the stack canary as a C string when controlling adjacent memory. If stack canary did not include a null byte, one could fill stack memory up until the stack canary, and then read it back to leak the whole canary.

Following is a simplified version of the `gets()` code example in figure 3.1.

```c
#include <stdio.h>

void function(void)
{
    char buf[32];

    gets(buf);
}

int main(void)
{
    function();

    return 0;
}
```

Let's take a look at the x86_64 assembly code when we compile without stack canaries.

```
    push   rbp
    mov    rbp,rsp
    sub    rsp,0x20
    lea    rax,[rbp-0x20]
    mov    rdi,rax
    mov    eax,0x0
    call   gets
    leave
    ret
```

And now with stack canaries enabled (using the `-fstack-protector` flag during compilation).

```
6aa:  55                    push   rbp
6ab:  48 89 e5              mov    rbp,rsp
6ae:  48 83 ec 30           sub    rsp,0x30
6b2:  64 48 8b 04 25 28 00  mov    rax,QWORD PTR fs:0x28
6b9:  00 00
6bb:  48 89 45 f8           mov    QWORD PTR [rbp-0x8],rax
6bf:  31 c0                 xor    eax,eax
6c1:  48 8d 45 d0           lea    rax,[rbp-0x30]
6c5:  48 89 c7              mov    rdi,rax
6c8:  b8 00 00 00 00        mov    eax,0x0
6cd:  e8 ae fe ff ff        call   580 <gets@plt>
6d2:  90                    nop
6d3:  48 8b 45 f8           mov    rax,QWORD PTR [rbp-0x8]
6d7:  64 48 33 04 25 28 00  xor    rax,QWORD PTR fs:0x28
6de:  00 00
6e0:  74 05                 je     6e7 <function+0x3d>
6e2:  e8 89 fe ff ff        call   570 <__stack_chk_fail@plt>
6e7:  c9                    leave
6e8:  c3                    ret
```

First of all, we can see that the code is a bit larger this time. At address `0x6b2`, a stack canary is fetched using the `fs` register. At `0x6bb`, the canary is stored as a local variable on the stack. This is a part of the function prologue. Then, at `0x6d3`, the canary is fetched from the stack and compared with the original canary in `fs`. If the canary has changed, for example as the result of stack overflow, the jump at `0x6e0` is not taken, and a call to `__stack_chk_fail()` is made. This function will abort the program and print an error message stating that stack corruption has been detected:

```
$ ./gets
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
*** stack smashing detected ***: <unknown> terminated
```

3.6 Other vulnerability classes

This section will describe other vulnerability classes, and how they can be exploited by an attacker to compromise a system. In section 3.6.2 we will discuss use-after-free vulnerabilities, and in section 3.6.3 we will discuss type confusion. We will start off by discussing format-string bugs in section 3.6.1, which are not so common today, but still an important part of the history of vulnerabilities.

3.6.1 Format string vulnerabilities

A *format string vulnerability* [59] arises when a user-controlled string is used as a format string in printf-style functions. Consider the prototype for `printf()`:

```c
#include <stdio.h>

int main(int argc, char *argv[])
{
        if (argc != 2)
                return 1;

        printf(argv[1]);

        return 0;
}
```

Figure 3.9: Format string vulnerability example

`int printf(const char *format, ...);`. The dots mean that this function takes a variable number of arguments. This is known as a *variadic function*.

When a function takes a variable amount of arguments, it is the caller's responsibility to set up registers and/or the stack properly so the function sees all the arguments. The callee has to know how many arguments are passed to it somehow. For printf-style functions, the amount of arguments correspond to the amount of *format specifiers* in the argument. These specifiers control how the argument is treated in the output. All format specifiers start with `%`, followed by one or more characters describing the format. A `%d` specifies that the next argument should be treated as an `int`, for example. Other specifiers include `%s` that treats the next argument as a string, `%lx` that treats the next argument as an `unsigned long` and displays it as a hexadecimal number, and so on. A problem arises when an attacker can supply the format string, and treat variables in registers or on the stack as arguments when they are not actually passed to the function.

This very easily leads to an information disclosure if the attacker specifies several `%p` arguments, for example. `%p` prints the next argument as a pointer, e.g. `0xdeadbeef`. For an information leak, the goal is to print an "argument" on the stack, or in a register, that contains a pointer to something that lets the attacker defeat ASLR. Consider the case where there is a pointer to `stdin` on the stack. If an attacker manages to print the address of `stdin` using a format string attack, the attacker can defeat ASLR. The reason an attacker can defeat ASLR in this case is because the `stdin` symbol is located at the same offset from the base of libc every time it is loaded. It is only the base address that is randomized with ASLR.

Let us take a look at an example. The program in figure 3.9 contains a format string vulnerability. If you try to compile this example with modern compilers, you will probably get a warning like this: `warning: format not a string literal and no format arguments`. This type of bug is not as common anymore, as compilers will complain if you attempt to pass something else than string literals to printf-style functions, making them easier to spot.

In addition to posing a risk for information leakage, format string bugs are actually very severe, as they also allow writing data to memory using the `%n` format specifier. In short, `%n` writes the amount of characters outputted by the function to the next parameter. An attacker can use this to corrupt memory and gain code execution.

3.6.2 Use-after-free vulnerabilities

A common class of software vulnerabilities are so-called *use-after-free* (UAF) vulnerabilities. As the name implies, this type of vulnerability occurs when we attempt to use memory after it has been freed.

The goal when exploiting UAF vulnerabilities is often to replace an existing object, which is then later used even though the object should have been freed. Usually, this type of bug can be exploited when you free some type of object that contains function pointers. If you can replace the object with one you control, you can control the function pointers, which again might let you take full control of the program flow.

As we will see in section 4.6, control-flow integrity is a security mitigation attempting to fix these types of exploitation techniques by restricting the number of valid call sites for a certain function pointer.

3.6.3 Type confusion

A type confusion vulnerability arises when an object of a type is unsafely treated as another type. Type confusion bugs are emerging as one of the most important attack vectors for C++ software [32]. Some examples of software written in C++ include web browsers like Google Chrome and Firefox, and their underlying JavaScript engines V8 and SpiderMonkey.

3.6.4 Heap vulnerabilities in general

UAF vulnerabilities and other heap-related vulnerabilities like heap buffer overflows are prevalent these days since the stack is often locked down with several mitigations in place. In [13] Bialek et. al. shows that heap overruns/overreads account for 13 % of all memory safety CVEs for Microsoft between 2015 and 2019. UAF vulnerabilities account for a significant 26 %, and non-adjacent heap out-of-bounds read or write accounts for 27 %. In total, that is 66 % of all memory corruption vulnerabilities for Microsoft in that period. Their research highlights the fact that heap vulnerabilities are very much alive even though there are many memory corruption mitigations in place to protect users.

3.6.5 Terminology

Now that we have a basic understanding of different vulnerabilities and how they can be exploited by an attacker, some more terminology can be introduced. These terms will be used in later chapters, and are particularly useful when reading articles on exploitation.

An exploit is often built using *primitives*, basic building blocks for an attacker. Commonly sought after primitives include *arbitrary read* and *arbitrary write* primitives. These allow an attacker to read arbitrary memory addresses and write to arbitrary memory addresses, respectively. An arbitrary write primitive is often known as a write-what-where primitive, which allows an attacker to write any value (what) to any address (where).

Chapter 4

Compilers and linkers

4.1 Introduction

In this chapter we will discuss compilers and linkers, and how the previously mention software defenses are implemented by modern compilers. We start with a general introduction in section 4.2. Then we briefly discuss gcc in section 4.3, and then move on to the LLVM project in section 4.4. In section 4.4.1 we will discuss in detail how different software defenses are implemented. We will then describe link-time optimization in section 4.5. After that we will dive into one of the central topics of this thesis, namely control-flow integrity. In section 4.6 we start by describing what control-flow integrity is, and then discuss how it is implemented. Finally, we will discuss some software defenses that are implemented in hardware in section 4.7.

4.2 General information

When compiling source code into an executable program, a lot of things happen behind the scenes that programmers normally do not have to think about. First, the code is passed to the preprocessor, which expands macros and replaces include statements with the full contents of the header, for example.

After the preprocessor has done its magic, the code is passed to the compiler. The compiler performs most of the heavy lifting in the compilation chain. The goal is to turn code into assembly, or another format that can easily be turned into assembly code. Such formats are usually known as an *intermediate representation*. The classical approach, however, is to turn C code into assembly, but we will see in later sections that clang does it a bit differently. After the compiler has produced assembly code, it is fed into an assembler, which translates assembly code into machine code. The output from an assembler is usually in the form of object files. An object file is not yet a full program, but it contains all the code for the original source file, symbol information, references to unresolved symbols, etc. The final step is to turn all our object files into an executable file. This step is performed by the linker.

When discussing executable files, we are referring to any type of executable file format. These formats differ a bit between the most popular operating systems. Linux uses the Executable and Linkable Format (ELF), Windows uses Portable Executables (PE), and macOS uses the Mach-O format. For our purposes, ELF files are the most relevant as we are only working with Linux.

4.3 GNU Compiler Collection

The GNU Compiler Collection (GCC) is a compiler system created by the GNU Project. The name gcc is often used to refer to the C compiler itself. Historically, gcc has been the only compiler used to compile the Linux kernel. It is still the compiler used to compile kernels for Ubuntu, Debian, Arch Linux, and virtually every single Linux distribution. gcc had its first release in 1987, thus it has a long history as a compiler.

We will use compiler features not available in gcc for later chapters, so there is no need to dive further into gcc here.

4.4 LLVM

LLVM started as a research project at the University of Illinois [44] with the "goal of providing a modern, SSA-based compilation strategy capable of supporting both static and dynamic compilation of arbitrary programming languages" [51]. Originally, LLVM was short for Low Level Virtual Machine, but the acronym is not used today. The LLVM project has grown into an umbrella project consisting of many sub-projects. These include a C/C++ compiler, a debugger, a C++ library, and others. See table 4.1 for a full list of projects.

name	description
LLVM core	core libraries, e.g. for code generation
clang	C/C++/Objective-C compiler
lldb	LLVM debugger
libc++ (ABI)	implementation of the C++ Standard Library
compiler-rt	code-generation
OpenMP	OpenMP runtime
polly	cache-locality optimizations
libclc	aims to implement the OpenCL standard
klee	symbolic virtual machine
lld	drop-in replacement for system linkers

Table 4.1: LLVM sub-projects

The projects that are interesting for this thesis are the LLVM core libraries, clang, and LLD. As written in table 4.1, LLD is a drop-in replacement for system linkers. LLD accepts the same command line arguments and linkers scripts as the standard GNU linkers on Linux. According to the LLVM documentation, the expectation is that LLD runs more than twice as fast as the GNU gold linker when linking a large program on a multicore machine. This is great for huge projects like the Linux kernel, where LLD will hopefully be able to speed up the linking process.

Clang is just one of many LLVM fontends, other languages that use LLVM include Rust, Swift, and Haskell. Although gcc has been around for a very long time, clang has attracted many users. Some examples of large projects that use clang are the Chrome and Firefox web browsers. FreeBSD has switched to clang/LLVM for some of its supported architectures. As of June 15, 2020, amd64, arm64, i386, armv7, and powerpc64 uses LLD to link both packages and the kernel. clang has replaced gcc as the standard compiler on the system as well.

4.4.1 Vulnerability mitigations

In chapter 3 we discussed software vulnerabilities and briefly how compilers can mitigate some of these vulnerabilities. In the following sections, we will look more into the standard security mitigations available today and how they are implemented.

4.4.2 ASLR/PIE

As we discussed in section 3.5.1, certain memory regions of programs are loaded at random addresses to thwart certain attacks. Most exploits require that an attacker know at least some of the program's layout. The stack, heap, and shared libraries will be loaded at different locations in memory. For the program to be loaded at a random location as well, it has to be a position independent executable (PIE). A PIE is constructed in such a way that it can be loaded anywhere in memory without relying on hard-coded memory addresses. This is easier on some architectures like x86_64 where we have what is known as RIP relative addressing. This means that the program can access memory relative to the current instruction pointer (`rip`). x86 (32-bit) does not have this feature and has to rely on other tricks to make this work.

4.4.3 RELRO

When a program wants to call a function that is located in a library, say for example `printf()`, the program has to get the address of this function somehow. This is handled by the dynamic linker on the system. The job of the dynamic linker is to load the executable into memory and resolve any dependencies on other libraries. This includes loading the required libraries into memory and resolving any function addresses that the program depends on in these libraries. The process of connecting a function symbol, like `printf()`, to an address is known as *relocation*. For ELF files we have two interesting sections pertaining to relocations. First we have the *global offset table* (GOT) and the *procedure linkage table* (PLT). There are other sections related to relocations as well, but these are the most interesting for our purposes. The PLT contains code stubs that jump to an address defined in the GOT.

For performance reasons, however, the symbols in the GOT table might not be resolved right away. Instead, the entries will point to a function that performs the actual resolving at runtime. To support this scheme, the GOT has to be writable. An attacker that has an arbitrary write primitive will be able to change GOT entries to point to anything else. Common ways to abuse this is to change functions that deal with user-controlled data into `system()` or other functions that can be used to execute arbitrary commands.

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

int main(int argc, char *argv[])
{
        char buf[32] = { 0 };

        if (argc == 2)
                strcpy(buf, argv[1]);

        printf("yay: %s\n", buf);

        return 0;
}
```

Figure 4.1: Simple `strcpy()` example

To protect against attacks on the GOT table, compilers support something known as *relocations read-only* (RELRO). When RELRO is enabled, the GOT table is only writable while the dynamic linker is resolving all the required functions. After relocation is finished, the section is marked as read-only, thus preventing an attacker from modifying the GOT.

4.4.4 NX

The *non-executable* (NX) bit marks a memory region as non-executable. This mitigation is usually known as *data execution prevention* (DEP) [57] on Windows. Traditionally, many memory regions were marked as executable even though they did not contain any code. For example the stack and the heap used to be marked as executable. An attacker could abuse the fact that these regions were both writable and executable at the same time to inject code into the stack or the heap and then divert execution to this location. The memory protections for different memory regions are controlled by the ELF headers. Normal regions of memory like program code is controlled using section headers. Today, the stack and heap are not executable by default, but can be made executable using the `PT_GNU_STACK` program header.

4.4.5 Fortify source

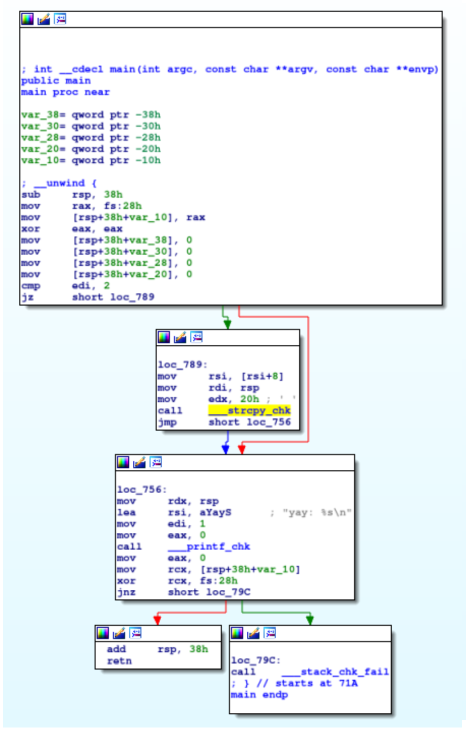
`FORTIFY_SOURCE` is a macro that provides basic protection from buffer overflows. Setting the macro to 1 or 2 enables buffer overflow checking for the following functions: `memcpy`, `mempcpy`, `memmove`, `memset`, `strcpy`, `stpcpy`, `strncpy`, `strcat`, `strncat`, `sprintf`, `vsprintf`, `snprintf`, `vsnprintf`, and `gets`.

When enabled, it checks the amount of bytes copied to make sure that buffers are not overflowed. If a buffer overflow is detected at runtime, an error message is printed and the program is aborted. To show how fortify source works, consider the source code example in figure 4.1.

What happens to the executable when we compile it with and without fortify source? Attempting to send more than 32 bytes in the first argument causes the program to crash if we do not have fortify source enabled. However, when fortify source is enabled, we get the following error message:

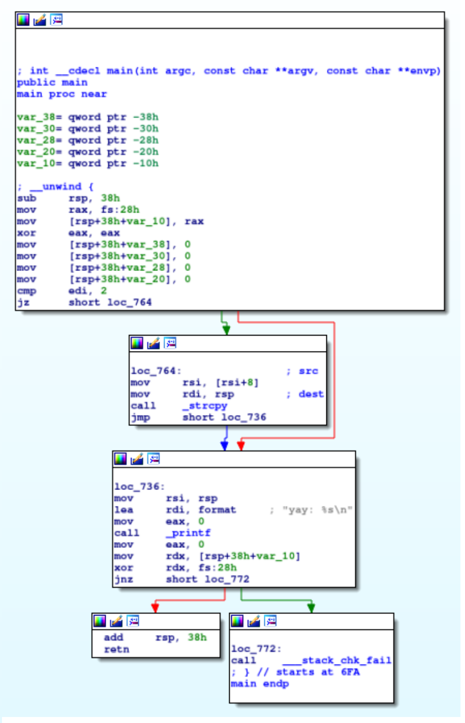
```
*** buffer overflow detected ***: ./main terminated
```

So how does the program detect this at runtime? Let us look at the control-flow graph of the program compiled with and without fortify source side by side to see what is different. Consider the two CFGs in figure 4.2. We can see that there is almost no difference between the two CFGs. However, if you look closely you can see that `strcpy()` has been replaced by `__strcpy_chk()`, and `printf()` has been replaced by `__printf_chk()`.



```
; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_38= qword ptr -38h
var_30= qword ptr -30h
var_28= qword ptr -28h
var_20= qword ptr -20h
var_10= qword ptr -10h

; __unwind {
sub     rsp, 38h
mov     rax, fs:28h
mov     [rsp+38h+var_10], rax
xor     eax, eax
mov     [rsp+38h+var_38], 0
mov     [rsp+38h+var_30], 0
mov     [rsp+38h+var_28], 0
mov     [rsp+38h+var_20], 0
cmp     edi, 2
jz      short loc_789

loc_789:
mov     rsi, [rsi+8]
mov     rdi, rsp
mov     edx, 20h ; ' '
call    ___strcpy_chk
jmp     short loc_756

loc_756:
mov     rdx, rsp
lea     rsi, aYayS      ; "yay: %s\n"
mov     edi, 1
mov     eax, 0
call    ___printf_chk
mov     eax, 0
mov     rcx, [rsp+38h+var_10]
xor     rcx, fs:28h
jnz     short loc_79C

 add     rsp, 38h
 retn

loc_79C:
call    ___stack_chk_fail
; } // starts at 71A
main endp
```

(a) CFG of program without fortify source



```
; int __cdecl main(int argc, const char **argv, const char **envp)
public main
main proc near

var_38= qword ptr -38h
var_30= qword ptr -30h
var_28= qword ptr -28h
var_20= qword ptr -20h
var_10= qword ptr -10h

; __unwind {
sub     rsp, 38h
mov     rax, fs:28h
mov     [rsp+38h+var_10], rax
xor     eax, eax
mov     [rsp+38h+var_38], 0
mov     [rsp+38h+var_30], 0
mov     [rsp+38h+var_28], 0
mov     [rsp+38h+var_20], 0
cmp     edi, 2
jz      short loc_764

loc_764:                        ; src
mov     rsi, [rsi+8]
mov     rdi, rsp                ; dest
call    _strcpy
jmp     short loc_736

loc_736:
mov     rsi, rsp
lea     rdi, format     ; "yay: %s\n"
mov     eax, 0
call    _printf
mov     eax, 0
mov     rdx, [rsp+38h+var_10]
xor     rdx, fs:28h
jnz     short loc_772

 add     rsp, 38h
 retn

loc_772:
call    ___stack_chk_fail
; } // starts at 6FA
main endp
```

(b) CFG of program with fortify source

Figure 4.2: `_FORTIFY_SOURCE` comparison.

The implementation of `__strcpy_chk()` can be seen in figure 4.3 [1].

Note that `__strcpy_chk()` actually takes one extra parameter compared to `strcpy()`, which is the length of the destination buffer. If the length of the source string exceeds the destination buffer length, the program prints out the error message we saw earlier.

Fortify source attempts to thwart format string attacks by performing more sanity checking of the format string. When positional arguments are used, the runtime verifies that all preceeding arguments are used. Positional arguments are constructed as follows: `%N$s`, where `N` is the position of the argument, and `s` can be any format specifier. Consider the following C code snippet:

[1]see the glibc source code: `debug/strcpy_chk.c`

```c
/* Copy SRC to DEST with checking of destination buffer overflow.  */
char *
__strcpy_chk (char *dest, const char *src, size_t destlen)
{
  size_t len = strlen (src);
  if (len >= destlen)
    __chk_fail ();

  return memcpy (dest, src, len + 1);
}
```

Figure 4.3: `__strcpy_chk()` implementation

```c
#include <stdio.h>

int main(void)
{
        printf("%2$d %1$s %4$x\n", "test", 123, 456, 0xbad);
        return 0;
}
```

The program will print: `123 test bad`.

Additionally, the `%n` format specifier can be used to write to memory. The `printf` man page explains like this [2]: "The number of characters written so far is stored into the integer pointed to by the corresponding argument." The man page also explains that this can be a security hole: "Code such as printf(foo); often indicates a bug, since foo may contain a % character. If foo comes from untrusted user input, it may contain %n, causing the printf() call to write to memory and creating a security hole." With fortify source, `%n` specifiers can only be used from read-only memory, thwarting attempts to inject these specifiers into an attacker-controlled format string. There have been attacks bypassing this protection, however [65].

4.4.6 Compiler mitigation summary

Table 4.2 summarizes the different compiler mitigations and what compiler flags are used to enabled or disable them.

mitigation	enabled by	disabled by
Stack cookie	`-fstack-protector`	`-fno-stack-protector`
NX	default	`-Wl,-z,execstack`
Fortify source	`-D_FORTIFY_SOURCE=1,2`	`-U_FORTIFY_SOURCE`
RELRO	`-Wl,-z,relro,-z,now`	`-Wl,-z,norelro`
PIE	`-fPIC -pie`	`-no-pie`

Table 4.2: Compiler mitigation summary

[2] `man 3 printf`

4.4.7 Summary

As the previous chapters show, there are many security mitigations available in compilers today. These help mitigate the impact of security vulnerabilities in software from increasingly advanced attacks. These mechanisms do not prevent exploitation of all memory corruption vulnerabilities, but they make it significantly harder for an attacker to succeed. In many cases an attacker needs several vulnerabilities to exploit a system. With ASLR an attacker often needs an information leak before exploitation of buffer overflows is possible, for example. In section 4.6 and 4.6.3 we will look at some relatively new mechanisms available in modern compilers.

4.5 Link-time optimization

In this section we will describe *link-time optimization* (LTO) and how it is implemented in gcc and clang [47]. LTO is a form of *interprocedural optimization* (IPO), where the goal is to optimize a program as a whole, instead of within a single function or single block of code. Traditionally a compiler like gcc or clang will compile every single C source file to object files. Then, the linker will combine them to produce an executable file, e.g an ELF file on Linux. With LTO, the linker will perform optimizations while looking at *all* of the object files and will attempt to optimize across object file boundaries. This makes it possible to perform optimizations such as removing functions that are exported in one object file, but is not actually used by any other code, i.e. dead code elimination. Other possibilities include inlining functions from one object file in another which will remove function call overhead at the cost of increasing function size. As we will see in section 4.6.1, LTO enables security features like CFI that need to analyze the whole program across translation units.

4.5.1 Link-time optimization in LLVM

This section describes how LTO is designed in LLVM. To support LTO, LLVM integrates tightly with the linker. The linker provides full transparency of symbols to the LLVM optimizer, which enables it to perform actions such as removing unused functions across object file boundaries.

There are two types of LTO in LLVM, the first is simply known as LTO or full LTO, and the second is known as ThinLTO [49]. ThinLTO was introduced in 2015, and has seen a number of improvements since then. Before we dive into the differences between ThinLTO, and what sort of advantages it brings, we will look at full LTO.

When using LTO, clang will emit LLVM bitcode instead of object files. LLVM will load and merge all bitcode files together to produce a single module. Then, interprocedural analyses and interprocedural optimizations are performed on this single, monolithic module. Since a monolithic module is created, LTO often requires a large amount of memory. This may be an issue for large projects. In addition, this process is quite slow. Every time a change is done to any source file in the project, the whole LTO process has to be performed again, significantly impacting incremental compilation performance.

The issues with memory requirements and incremental builds meant that a new way of doing LTO was desired. ThinLTO was designed for this purpose.

ThinLTO does not load everything into a single monolithic module. C code is processed as usual by clang and initial optimizations are performed. A summary of each function is generated, and only this summary info is linked into a giant index called *thin-link*. Optimizations and code generation is fully-parallel, unlike with full LTO.

4.6 Control-flow integrity

In this section, we will describe *control-flow integrity* (CFI) [1]. CFI aims to limit the *control-flow graph* (CFG) of a program to known destinations computed at compile time. CFI considers edges in the CFG of a program, and enforces control-flow to follow known edges in the CFG. CFI attemps to reduce the impact of memory corruption bugs by limiting control-flow to a certain set of valid locations, which means that attempting to overwrite function pointers to point inside the body of a function does not work, for example. As there are (usually) no situations where one would call an offset into a function, it is not a valid destination for control-flow instructions.

There are two types of CFI: *forward-edge* and *backward-edge*. Forward-edge CFI protects forward edges in the CFG, for example indirect function calls through function pointers. Backward-edge CFI protects edges going backwards in the CFG. For example returning from a function using the `ret` instruction on x86_64 or arm64.

In their paper, Tice et. al [73] describes how CFI was implemented in gcc and LLVM. The implementation for gcc is called Virtual-Table Verification (VTV). It only protects *virtual calls*, thus only works for C++ programs. The CFI implementation for LLVM is called Indirect Function-Call Checks (IFCC) and protects all indirect function calls, including those not made through vtables.

4.6.1 Forward-edge control-flow integrity in clang

Next, we will discuss how forward-edge CFI is implemented in clang. Clang's CFI supports many schemes for protecting different types of function calls. Many of these supported calls are only relevant for C++, so they are only mentioned here for completeness. Table 4.3 lists all the supported schemes. One can use the command line arguments `-fsanitize=<NAME>` to enable a scheme, or `-fno-sanitize=<NAME>` to disable it when compiling.

Available CFI Schemes

compiler flag	description
`cfi-cast-strict`	Enables strict cast checks
`cfi-derived-cast`	Base-to-derived cast to the wrong dynamic type
`cfi-unrelated-cast`	Cast from `void *` or another unrelated type to the wrong dynamic type
`cfi-nvcall`	Non-virtual call via an object whose vptr is of the wrong dynamic type
`cfi-vcall`	Virtual call via an object whose vptr is of the wrong dynamic type
`cfi-icall`	Indirect call of a function with wrong dynamic type
`cfi-mfcall`	Indirect call via a member function pointer with wrong dynamic type

Table 4.3: CFI schemes

It is possible to enable all the schemes by passing the `-fsanitize=cfi` flag to clang when compiling, or one could narrow down the set of schemes by using `-fsanitize` or `-fno-sanitize` to enable or disable some of them, respectively.

Since the Linux kernel is written in C and assembly, the C++ implementation is not as relevant for this thesis. The only type of function call we are interested in are indirect function calls, which are performed using function pointers in C. At each call site, an extra check verifies that the function being called is a valid function that matches the function pointer's signature. Consider the code example in figure 4.4.

In the code we have a struct named `ops` that contains a single function pointer. The signature matches the `foo()` and `baz()` functions. Note that it does not match the function signature of `bar()`, since this function takes an `int` argument, while the others do not. When the program is started with no arguments, the `foo()` function is run. If there is one argument, `bar()` is used instead. Finally, if there are two arguments, `baz()` will be used. CFI should be able to catch the function signature mismatch at runtime. The following snippet shows what happens when the program is run with zero, one, and two arguments:

```
$ ./cfi
op.func() = 1337
$ ./cfi a
cfi.c:35:29: runtime error: control flow integrity check for type
             'int (void)' failed during indirect function call
(cfi+0x234540): note: bar defined here
$ ./cfi a b
op.func() = -559038737
```

The functions with correct prototypes succeeded, as expected, and the function signature mismatch was caught at runtime. We will now discuss how the CFI checks are performed at runtime. When compiling, clang will turn C code into something known as LLVM *intermediate representation*, or *IR* for short, before any assembly code is emitted. IR is an assembly-like representation of the code used internally in the compiler. One advantage of using IR is that compilers can generate code for many different languages as long as they are translated to the same IR. Further analysis and transformation from IR to machine code can be reused.

IR can be emitted for several phases of compilation, which we will discuss further later in this section. In figure 4.5 we can see IR [3] for for the code in figure 4.4 during the preoptimization phase. In this phase, no optimizations have been done by the compiler yet. The code has been simplified slightly, and all omitted code is replaced with `[...]`. In the IR, indirect function calls are replaced with the LLVM instrinsic function `llvm.type.test`. As the name implies, this intrinsic checks the type of the function.

Following is the type test extracted from the IR:

```
  %25 = call i1 @llvm.type.test(i8* %24, metadata !"_ZTSFivE")
```

The most interesting part of the type test is the final argument, `"ZTSFivE"`. This is the mangled type name the code expects the function pointer signature

[3]produced using `-Wl,-save-temps` flag to store bitcode during compilation and then disassembling using `llvm-dis`

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

int foo(void)
{
        return 1337;
}

int bar(int a)
{
        return a * 42;
}

int baz(void)
{
        return 0xdeadbeef;
}

struct ops {
        int (*func)(void);
};

int main(int argc, char *argv[])
{
        struct ops op;

        (void)argv;
        memset(&op, 0, sizeof(op));
        if (argc == 2)
                op.func = (int (*)(void))bar;
        else if (argc == 3)
                op.func = baz;
        else
                op.func = foo;
        printf("op.func() = %d\n", op.func());

        return 0;
}
```

Figure 4.4: CFI code example

to match. The name can be decoded using the `c++filt` tool: [4] `int ()`. That is, a function that returns `int` and takes no arguments. This matches the function pointer signature from the C code:

```
struct ops {
        int (*func)(void);
};
```

During the IPO phase [5], LLVM will perform lowering of the `llvm.type.test` function into its actual implementation. The implementation looks a bit strange in assembly code, so it is worth to spend some time explaining it here. First of all, the functions with matching prototypes are laid out after each other in a jump table. For the previous C code example in figure 4.4, the jump-table in pseudo-C would look something like this:

```
static void *jump_table_int_void[] = { foo, baz };
static void *jump_table_int_int[]  = { bar };
```

The jump table corresponding to the `int ()` function signature contains two entries: `foo()` and `baz()`. The jump table for the type `int (int)` contains only one entry: `bar()`. These jump tables make it easier to quickly verify whether a function belongs to a particular set of function signatures. The compiler makes sure that only functions with the same signature are laid out consecutively.

The first step when lowering the call is to calculate the offset into the jump table. Then, two things need to be checked to verify that the call is valid. First, the offset has to fall within range of the jump table entries for this function type. Second, we need to check that the function is properly aligned. Recall that on x86_64 instructions do not need to be aligned on any particular boundary, as the instruction length is variable. On arm64, all instructions have to be aligned on a 4-byte boundary since all instructions are 4 bytes long.

On x86_64, each jump table entry is aligned to 8 bytes, on arm64 they are aligned to 4 bytes. The reason lies in the implementation of the table entries. For x86_64 a jump table entry is constructed using a `jmp` instruction, which is (usually) 5 bytes in size. The instruction opcode is `e9`, followed by a 4-byte offset relative to the next instruction. To align the next entry in the jump table to 8 bytes, `int3` instructions are inserted as padding. In figure 4.6 we have the jump table entries for `foo()` and `baz()` as outputed by `objdump` [6]. The `int3` instructions will cause a trap signal, effectively aborting the program if program flow is intentionally or unintentionally redirected to the padding area.

[4] `echo "_ZTSFivE" | c++filt`

[5] See `LowerTypeTestsModule::lowerTypeTestCall()` in `lib/Transforms/IPO/LowerTypeTests.cpp`

[6] `objdump -D cfi`

```
; Function Attrs: noinline nounwind optnone uwtable
define hidden i32 @main(i32, i8**) #0 !type !8 !type !9 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  %6 = alloca %struct.ops, align 8
  store i32 0, i32* %3, align 4
  store i32 %0, i32* %4, align 4
  store i8** %1, i8*** %5, align 8
  %7 = load i8**, i8*** %5, align 8
  %8 = bitcast %struct.ops* %6 to i8*
  call void @llvm.memset.p0i8.i64(i8* align 8 %8, i8 0, i64 8, i1 false)
  %9 = load i32, i32* %4, align 4
  %10 = icmp eq i32 %9, 2
  br i1 %10, label %11, label %13

11:                                               ; preds = %2
  %12 = getelementptr inbounds %struct.ops, %struct.ops* %6, i32 0, i32 0
  store i32 ()* bitcast (i32 (i32)* @bar to i32 ()*), i32 ()** %12, align 8
  br label %21

13:                                               ; preds = %2
  %14 = load i32, i32* %4, align 4
  %15 = icmp eq i32 %14, 3
  br i1 %15, label %16, label %18

16:                                               ; preds = %13
  %17 = getelementptr inbounds %struct.ops, %struct.ops* %6, i32 0, i32 0
  store i32 ()* @baz, i32 ()** %17, align 8
  br label %20

18:                                               ; preds = %13
  %19 = getelementptr inbounds %struct.ops, %struct.ops* %6, i32 0, i32 0
  store i32 ()* @foo, i32 ()** %19, align 8
  br label %20

20:                                               ; preds = %18, %16
  br label %21

21:                                               ; preds = %20, %11
  %22 = getelementptr inbounds %struct.ops, %struct.ops* %6, i32 0, i32 0
  %23 = load i32 ()*, i32 ()** %22, align 8
  %24 = bitcast i32 ()* %23 to i8*, !nosanitize !10
  %25 = call i1 @llvm.type.test(i8* %24, metadata !"_ZTSFivE"), [...]
  br i1 %25, label %28, label %26, !prof !11, !nosanitize !10

26:                                               ; preds = %21
  %27 = ptrtoint i8* %24 to i64, !nosanitize !10
  call void @__ubsan_handle_cfi_check_fail_abort([...])
  unreachable, !nosanitize !10

28:                                               ; preds = %21
  %29 = call i32 %23()
  %30 = call i32 (i8*, ...) @printf([...])
  ret i32 0
}
```

Figure 4.5: LLVM IR excerpt

```
0000000000234620 <foo>:
  234620:       e9 0b ff ff ff          jmpq   234530 <foo.cfi>
  234625:       cc                      int3
  234626:       cc                      int3
  234627:       cc                      int3

0000000000234628 <baz>:
  234628:       e9 23 ff ff ff          jmpq   234550 <baz.cfi>
  23462d:       cc                      int3
  23462e:       cc                      int3
  23462f:       cc                      int3
```

Figure 4.6: CFI jump table disassembly

On arm64, every instruction is 4 bytes, so only a `b` (branch) instruction is used. A clever trick is used by LLVM to check that the jump table entry is in-bounds and that the alignment is correct. First, the offset into the jump table is calculated by subtracting the start of the jump table for the relevant type signature from the actual jump table entry we want to call. This offset is then rotated right by $\log_2(alignment)$ bits, resulting in a valid index into the table. Looking at the addresses in figure 4.6, we can perform the calculation as see that it makes sense. Consider a situation where we want to call `baz()`. First we need the address of the function, which is `0x234628`. Then we need the alignment in bytes, which we recall is 8 bytes on x86_64. Finally we need the address of the jump table, which in this case starts at the same address as `foo()`, `0x234620`. We can then calculate the offset into the table: $\texttt{0x234628} - \texttt{0x234620} = 8$. Then we rotate the result: $8 \text{ ror } \log_2(8) = 8 \text{ ror } 3 = 1$. The result makes sense since `baz()` is entry number one in the zero-indexed table. Now imagine that we attempt to call an invalid address, for example `0x234629`. Again we calculate the offset: $\texttt{0x234629} - \texttt{0x234620} = 9$. The offset is now one more than the previous one. Now what happens when we rotate 9 right by 3? Since we are dealing with 64-bit integers, we get the huge number `0x2000000000000001`. The reason for this huge number is because 9 is not properly aligned on an 8-byte boundary, which means that some of its lower bits will propagate to the upper bits of the result. All the code needs to do to verify the table index is to compare it against the number of table entries. For unaligned function addresses the index will be way out of bounds, and for aligned addresses it will work just like a normal comparison operation.

The CFI checking can be a lot more complicated when we are dealing with C++. In C++ classes, we have something known as virtual methods. These can be inherited from parent classes, and can be overridden by classes that inherit them. Calling a virtual function is known as a *virtual call*. An important piece of every C++ class is its virtual method table, or vtable for short. The vtable is basically a list of all the functions a class can use. When we are dealing with C functions, the valid targets are laid out consecutively in a jump table with a static offset between each function. Vtables are laid out differently, however, which makes it more complicated to check for valid function calls. A bit vector is created to represent the class layout. These bit vectors are stored as byte arrays in the actual program. To verify that a call is correct, an index in the byte array can be fetched and tested using a bit mask.

The need for bit vectors is usually eliminated for indirect function calls, which makes the checks somewhat simpler compared to virtual calls. Loading from and checking the values in the bit vector is thus not included in the code examples we will be looking at.

The LLVM IR we saw in figure 4.5 corresponded to the preoptimization phase of compilation. During the next phases, the `llvm.type.test` intrinsic is lowered to IR more resembling the final assembly code output. In figure 4.7 we can see the LLVM IR after lowering the type test.

```
21:
  %22 = getelementptr inbounds %struct.ops, %struct.ops* %6, i32 0, i32 0
  %23 = load i32 ()*, i32 ()** %22, align 8
  %24 = bitcast i32 ()* %23 to i8*, !nosanitize !10
  %25 = ptrtoint i8* %24 to i64
  %26 = sub i64 %25, ptrtoint (void ()* @.cfi.jumptable to i64)
  %27 = lshr i64 %26, 3
  %28 = shl i64 %26, 61
  %29 = or i64 %27, %28
  %30 = icmp ule i64 %29, 1
  br i1 %30, label %33, label %31, !prof !11, !nosanitize !10

31:                                                   ; preds = %21
  %32 = ptrtoint i8* %24 to i64, !nosanitize !10
  call void @__ubsan_handle_cfi_check_fail_abort([...])
  unreachable, !nosanitize !10
  33:                                                   ; preds = %21
  %34 = call i32 %23()
```

Figure 4.7: LLVM IR excerpt

Note that the type test is now turned into a sequence of `sub`, `lshr`, `shl`, and `or` instructions more resembling normal assembly code. The actual assembly code does not look that different from LLVM IR actually. A comparison between x86_64 and arm64 assembly code can be seen in figure 4.8. The code is almost exactly the same size, only differing by one instruction. The assembly code was created using IDA Pro [33] and cleaned up for clarity. Also note that `abort()` is not actually called; the real code calls `__ubsan_handle_cfi_check_fail_abort()`. This function is part of the compiler-rt project, which contains several runtime libraries. Looking at the function name we can see that it is part of the *UBSan*, or *UndefinedBehaviorSanitizer* [50] runtime.

The `@PAGE`/`@PAGEOFF` syntax may seem a bit strange. This is the way IDA Pro displays it, which is the tool used to produce these assembly listings. The `adrp` instruction fetches the address of a 4KB page at a PC-relative offset. An `add` instruction can then be used to add the offset of the wanted symbol to the page address. This pattern is used since we cannot encode full addresses (4 or 8 byte) in instructions since all instructions are only 4 bytes in size on arm64. The `adr`/`add` instructions are used to work around this limitation.

When LLVM generates the jump tables, their entries use the original symbol

```
adrp    x8, foo@PAGE
add     x8, x8, foo@PAGEOFF
adrp    x9, bar
add     x9, x9, bar
str     x9, [sp,#8]
ldr     x1, [sp,#8]
sub     x8, x1, x8
lsr     x9, x8, #2
orr     x8, x9, x8,lsl#62
cmp     x8, #1
b.ls    loc_9D0
call    abort
loc_9D0:
mov     w0, #4
blr     x1
```

(a) arm64 CFI check

```
mov     rax, offset bar
mov     [rbp-16], rax
mov     rsi, [rbp-16]
mov     rax, offset foo
mov     rcx, rsi
sub     rcx, rax
mov     rax, rcx
shr     rax, 3
shl     rcx, 3Dh
or      rax, rcx
cmp     rax, 1
jbe     short loc_2354AD
call    abort
loc_2354AD:
mov     edi, 4
call    rsi
```

(b) x86_64 CFI check

Figure 4.8: Comparison between CFI checks on arm64 and x86_64

names from functions. Thus, the `foo()` symbol from the previous examples used to point to the original function, but now points to the jump table instead. Every function that is redirected through the jump table get a `.cfi` suffix. `foo()` turns into `foo.cfi()`, for example.

Since CFI instruments every single indirect function call (unless told not to do so explicitly [7]), it is interesting to consider how much overhead is generated for each instrumented call site. In figure 4.9 we see a comparison between a normal call and a call protected with CFI for arm64.

We can see that there are a lot of instructions added to the CFI call. Including the two instructions not shown in the listing used to fetch the argument to the `abort()` function, 9 instructions have been added. This results in an increase of $4 \times 9 = 36$ bytes. In addition, a jump table entry is constructed, adding another 4 bytes. In figure 4.10 we have the same comparison for x86_64.

There are a lot of instructions added here as well. For this example, 11 instructions are added in the CFI example. It is not as easy to calculate the size overhead here, since instruction size varies on x86_64 and we cannot simply count the number of instructions added. The example without CFI enabled consists of 30 bytes, while the example with CFI consists of 76 bytes. This results in a 153.33 % increase in size, compared to a 166.67 % increase for arm64. These numbers may not be the same for every situation, depending on compiler optimizations and other factors, but they highlight the fact that CFI adds some extra size to the code.

4.6.2 Cross-DSO CFI

LLVM also supports something known as *cross-DSO* CFI. DSO is short for *dynamic shared object*, which is commonly known as a shared library. If cross-DSO

[7] using the `__no_sanitize__("cfi")` attribute.

```
adrp    x8, bar@PAGE
add     x8, x8, bar@PAGEOFF
str     x8, [sp,#8]
ldr     x8, [sp,#8]
mov     w0, #4
blr     x8
```

(a) Normal arm64 call

```
adrp    x8, foo@PAGE
add     x8, x8, foo@PAGEOFF
adrp    x9, bar@PAGE
add     x9, x9, bar@PAGEOFF
str     x9, [sp,#8]
ldr     x1, [sp,#8]
sub     x8, x1, x8
lsr     x9, x8, #2
orr     x8, x9, x8,lsl#62
cmp     x8, #1
b.ls    loc_9D0
call    abort
loc_9D0:
mov     w0, #4
blr     x1
```

(b) CFI arm64 call

Figure 4.9: Comparison between calls with and without CFI on arm64

```
mov    rdi, offset bar
mov    [rbp-24], rdi
mov    edi 4
mov    [rbp-28], eax
call   qword ptr [rbp-24]
```

(a) Normal x86_64 call

```
mov     rax, offset bar
mov     [rbp-16], rax
mov     rsi, [rbp-16]
mov     rax, offset foo
mov     rcx, rsi
sub     rcx, rax
mov     rax, rcx
shr     rax, 3
shl     rcx, 3Dh
or      rax, rcx
cmp     rax, 1
jbe     short loc_2354AD
call    abort
loc_2354AD:
mov     edi, 4
call    rsi
```

(b) x86_64 CFI check

Figure 4.10: Comparison between calls with and without CFI on x86_64

CFI is enabled, control flow checking will be enforced across DSO boundaries, meaning that if you attempt to call functions in another library with the wrong function signature the call will be blocked at runtime.

4.6.3 Backward-edge control-flow integrity in clang

A fairly new vulnerability mitigation implemented in LLVM is *ShadowCallStack* (SCS) [48]. This mechanism attempts to mitigate backward-edge control flow hijacking by protecting the return address on the stack. To accomplish this, the return values are loaded from a separate stack known as the shadow stack. This stack, however, does not mirror all the data on the normal stack. It contains an array of return addresses. Although a powerful mitigation, the x86_64 implementation was thrown out of LLVM because of performance and security issues. The current implementation is only supported on aarch64. Intel has planned another security feature called Control-flow Enforcement Technology [36] [14] [52] [21] which is implemented in hardware. Since Intel will add backward-edge protection in hardware, and because the ShadowCallStack implementation for x86_64 caused a lot of performance overhead it was removed in LLVM 9.0.

SCS requires runtime support, for example in the C standard library for normal programs, or the Linux kernel for kernel drivers. Support for SCS is included in Android's libc, named bionic. The first implementation [8] allocated a region for the shadow stack, and placed it inside this region at a known offset. The current implementation [9], however, puts the shadow stack at a random offset inside the shadow stack region. The shadow stack is both readable and writable, but the rest of the region is mapped without any access rights. Thus, any attempt to access memory before or after the shadow stack results in a crash. This provides protection for both stack underflows and overflows. Randomizing the location of the shadow stack further complicates attacks targeting the shadow stack. See figure 4.11 for the SCS implementation in bionic.

The SCS code is part of the code for *POSIX threads* [55], or pthreads. Whenever a thread is created, or during initialization of a program's main thread, a new shadow stack is created using `__init_shadow_call_stack()`. At the end of the function we can see that the return value, `scs`, is placed in `x18` using inline assembly. Clang deliberately uses `x18` to store the shadow stack, which means that no other code should use this register. To ensure that no code uses `x18`, the `-ffixed-x18` compiler flag can be used.

Let us examine how SCS looks like in a compiled program. The C code example can be found in figure 4.12. The assembly code output of `main()` can be found in figure 4.13. Note that the `foobar()` function does not change when compiled with SCS since it is a leaf function, and thus does not store the return address on the stack in the prologue (see section 2.6.1 for more information on leaf functions).

The assembly code output is very similar with and without SCS. The only difference is that the code with SCS enabled stores the return address (`x30`) into the shadow stack at the start of the function, and fetches in again at the end. Enabling SCS adds two instructions, or 8 bytes of code, to every non-leaf function in the program.

[8]see commit `808d176e7e0dd727c7f929622ec017f6e065c582`

[9]commit `c2a67121c747b5d1f43010b5a78032a95a722631`

```
static void __init_shadow_call_stack(pthread_internal_t* thread __unused) {
#ifdef __aarch64__
  // Allocate the stack and the guard region.
  char* scs_guard_region = reinterpret_cast<char*>(
      mmap(nullptr, SCS_GUARD_REGION_SIZE, 0, MAP_PRIVATE | MAP_ANON, -1, 0));
  thread->shadow_call_stack_guard_region = scs_guard_region;

  // The address is aligned to SCS_SIZE so that we only need to store the lower log2(SCS_SIZE) bits
  // in jmp_buf.
  char* scs_aligned_guard_region =
      reinterpret_cast<char*>(align_up(reinterpret_cast<uintptr_t>(scs_guard_region), SCS_SIZE));

  // We need to ensure that [scs_offset,scs_offset+SCS_SIZE) is in the guard region and that there
  // is at least one unmapped page after the shadow call stack (to catch stack overflows). We can't
  // use arc4random_uniform in init because /dev/urandom might not have been created yet.
  size_t scs_offset =
      (getpid() == 1) ? 0 : (arc4random_uniform(SCS_GUARD_REGION_SIZE / SCS_SIZE - 1) * SCS_SIZE);

  // Make the stack readable and writable and store its address in register x18. This is
  // deliberately the only place where the address is stored.
  char *scs = scs_aligned_guard_region + scs_offset;
  mprotect(scs, SCS_SIZE, PROT_READ | PROT_WRITE);
  __asm__ __volatile__("mov x18, %0" ::"r"(scs));
#endif
}
```

Figure 4.11: ShadowCallStack support in bionic libc

```
#include <stdio.h>

int foobar(void)
{
        return 0xbadc0de;
}

int main(void)
{
        return foobar() + 0x1337;
}
```

Figure 4.12: ShadowCallStack code example

```
main:
sub  sp, sp, #32
stp  x29, x30, [sp, #16]
add  x29, sp, #16
mov  w8, #4919
stur wzr, [x29, #-4]
str  w8, [sp, #8]
bl   foobar
ldr  w8, [sp, #8]
add  w0, w0, w8
ldp  x29, x30, [sp, #16]
add  sp, sp, #32
ret
```

(a) ShadowCallStack disabled

```
main:
sub  sp, sp, #32
str  x30, [x18], #8
stp  x29, x30, [sp, #16]
add  x29, sp, #16
mov  w8, #4919
stur wzr, [x29, #-4]
str  w8, [sp, #8]
bl   foobar
ldr  w8, [sp, #8]
add  w0, w0, w8
ldp  x29, x30, [sp, #16]
ldr  x30, [x18, #-8]!
add  sp, sp, #32
ret
```

(b) ShadowCallStack enabled

Figure 4.13: Assembly code with ShadowCallStack disabled/enabled

4.7 Hardware alternatives

The implementations we have discussed so far are all implemented in software. Similar technologies in hardware are currently being worked on, most notably Intel's *Control-flow Enforcement Technology* [36] (CET) for x86_64 and Qualcomm's *Pointer Authentication* [67] (PAC) for ARMv8.3. We will discuss CET in section 4.7.1, and PAC in section 4.7.2.

4.7.1 CET

CET provides the following capabilities according to Intel: Shadow Stack, and indirect branch tracking. A shadow stack is a second stack used exclusively for control transfer operations. When enabled the `call` instruction pushes the return address both to the normal stack and to the shadow stack. When the function returns later, the `ret` instruction will pop the return address from both stacks and compare them. In the event that they do not match, an exception is raised. Shadow stacks should be properly protected so that they are not easy targets for attackers with an arbitrary read/write primitive. For CET, the shadow stack is separate from the normal stack and is only used to store control transfer information. It is protected using page table protections. Thus, the shadow stack is not directly writable by software. The stack is only accessed by control transfer instructions (like `call` and `ret`) and shadow stack management instructions. The other feature, indirect branch tracking, includes a new instruction called `endbranch` used to mark valid jump target addresses of indirect calls and jumps in the program. While the ShadowCallStack on arm64 uses the x18 register as a pointer to the shadow stack, CET introduces support for the shadow stack pointer (SSP) register.

Indirect branch tracking changes the behavior of `jmp` and `call` instructions. The CPU implements a state machine to track indirect branch instructions. If the following instruction after an indirect jmp or call is not an `endbranch` instruction, the control-flow transfer was not valid. In other words, `endbranch` is used to mark valid indirect call targets in the program. To provide backward compatibility with older CPUs that do not support CET, the `endbranch` instruction is encoded such that it is treated as a `nop` on these CPUs.

CET is not yet available in hardware, but support for it has been enabled in GCC 8, binutils 2.31, and glibc 2.28, and added to LLVM in december 2019. There is also a patchset available for adding support to the Linux kernel [81], but it has not been upstreamed yet.

4.7.2 PAC

PAC raises the bar for attackers trying to modify protected pointers in memory. In this section we will describe how PAC is designed and what protection it provides.

PAC takes advantage of the fact that the address space in 64-bit architectures is less than 64-bits. The actual number of bits depends on the platform, but usually 55 or 48 bits are actually used. This means that 64-bit pointers have unused bits in the upper part of the pointer. These unused bits can be used to store a pointer authentication code. The PAC is inserted into each protected pointer before writing it to memory, and verified before it is used. Several new instructions are introduced to *sign* and *authenticate* pointers.

PAC uses a cryptographically strong algorithm to authenticate pointers named QARMA [8]. QARMA is “a new family of lightweight tweakable block ciphers” [67]. A 128-bit key is used together with a context value to sign pointers. The context is useful to separate different classes of pointers, for example stack pointers and function pointers.

In section 3.5.2 we discussed how return addresses on the stack are protected using a stack cookie. This mitigation adds some overhead to every function prologue and epilogue since the cookie first have to be stored on the stack, and then validated before returning from the function. With PAC, a single instruction can be used to tag and verify the link register (LR), which is where the return addresses are stored on arm64. The instruction `PACIASP` is used to protect LR, and `AUTIASP` is used to verify it.

If pointers used for indirect function calls are signed using PAC, a form of CFI can be implemented which makes sure that no invalid pointers are used for function calls. Different contexts can be used to group function pointers together and provide something similar to what clang does for its CFI implementation, which we discussed in section 6.3.2.

Basic support for PAC exists in the Linux kernel, but there is no real usage yet. Apple’s mobile operating system iOS has support for PAC, and it is enabled on certain iPhones [10]. It will be interesting to see how support for PAC on Linux develops over the next few years.

Chapter 5

Linux kernel internals

5.1 Introduction

This chapter will go into a bit more detail on the Linux kernel, before delving into what sort of optimization we can do using standard compilers and linkers. First, we will talk more about the different releases of the kernel in section 5.2, then move on to the Linux kernel's build system in section 5.3. Finally we will discuss different security features commonly enabled in the Linux kernel in section 5.4.

5.2 Linux kernel versions

Many versions of the Linux kernel is in active use. These can be divided into the following categories:

Prepatch

These releases are often referred to as ”RC”, or release candidate, kernels. They are not yet ready for use in stable releases, and must be compiled from source. This is where new features can be tested before the end up in a normal stable release and can be used by users that aren't developers.

Mainline

Mainline is where all the new features from the rc releases are introduced. New mainline releases come out every 2-3 months.

Stable

A mainline release is considered stable after it has been released. Bug fixes for a stable release is backported from the mainline tree.

Longterm

Several ”longterm maintenance” kernel releases are provided for the purpose of backporting bugfixes to older releases. Longterm releases usually only see important bugfixes backported to them. However, new minor releases are relatively frequent, while new major releases are infrequent.

Following is a table, accurate as of January 30th 2020, of the current longterm release kernels and their maintainers [2].

Longterm release kernels

Version	Maintainer	Released	Projected EOL
5.4	Greg Kroah-Hartman & Sasha Levin	2019-11-24	Dec, 2021
4.19	Greg Kroah-Hartman & Sasha Levin	2018-10-22	Dec, 2020
4.14	Greg Kroah-Hartman & Sasha Levin	2017-11-12	Jan, 2020
4.9	Greg Kroah-Hartman & Sasha Levin	2016-12-11	Jan, 2023
4.4	Greg Kroah-Hartman & Sasha Levin	2016-01-10	Feb, 2022
3.16	Ben Hutchings	2014-08-03	Apr, 2020

Table 5.1: Longterm release kernels

5.3 Build system

This section will describe the build system employed by the Linux kernel. At the core of the Linux kernel build system is the kbuild infrastructure. Kbuild helps the kernel deal with the complex structure of the code base, in particular with all the different configuration options and architectures available. The kernel is configured using a top-level .config file. This file allows a user to turn on and off different options when compiling. Some examples of options include the page size, what hardware should be supported, and what architecture we are compiling for. These configuration options are prefixed with `CONFIG_` and ends with an equals sign and then the value for this particular option. The two most common values are `y` and `m`. `y` is used when you want to enable an option, for example to enable kernel debugging through the `CONFIG_KGDB` option. `m` is used when you want to compile something as a loadable kernel module instead of integrating it as a part of the monolithic kernel. To disable an options, `n` is used. For example `CONFIG_OVERLAY_FS=n` to disable overlay fs support. When discussing configuration options in future sections, the `CONFIG` prefix will be omitted for brevity. `KGDB` will be used instead of `CONFIG_KGDB`, for example.

In the top-level and arch specific directories you have normal Makefiles. Most other folders, however, have kbuild makefiles. These makefiles carry out commands passed down from above. The top-level Makefile is responsible for building vmlinux, which is the kernel image, and all modules.

There are several default make targets that make it easier to customize the kernel build process. To change configuration options using a GUI there is the `make menuconfig` option which lets users turn on and off options using an ncurses menu. There are several targets that help turn on and off large amounts of configuration options. For example, we have the `allmodconfig` target that sets as many configuration options to `m` as possible, which means that it will attempt to compile in support for as many modules as possible. There is also the similar `allyesconfig` which will build everything directly into the kernel instead of having the code as loadable kernel modules.

5.4 Linux kernel security

In section 4.4.1 we discussed different security mitigations provided by compilers. These protections are common for most software in user space as well as kernel space. There are some interesting security mitigations that are special for the Linux kernel, however. In the following subsections we will describe some of these to highlight some of the steps taken to make the Linux kernel more secure. These security features are controlled by configuration options and may not be enabled for certain distros or Android devices. Although most of these are not directly relevant for CFI, they highlight the need for a diverse set of different security mechanisms in a modern operating system kernel.

5.4.1 KASLR

As mentioned in section 3.4 we have a protection known as kernel address space layout randomization (KASLR). This is the kernel version of ASLR, which we described in section 4.4.2.

5.4.2 SMEP/PXN

Supervisor Mode Execution Prevention (SMEP) and *Privileged Execute Never* (PXN) are CPU features, for x86_64 and arm64 respectively, that prevents the kernel from executing code located in user space. Without these protections, an attacker can redirect execution from the kernel into user space, where the attacker potentially has full control over the memory layout. This type of attack is known as ret2usr [37]. On x86_64, SMEP is enabled by setting bit 20 in the CR4 register. On arm64, PXN is controlled through translation table entries.

5.4.3 SMAP/PAN

Supervisor Mode Access Prevention (SMAP) and *Privileged Access Never* (PAN) are CPU features for x86_64 and arm64 respectively that can prevent access to unprivileged data. It is used to make sure that the kernel cannot directly access memory in user space. To access memory in user space, the kernel has to use functions designed for this purpose, like `copy_from_user()`. On x86_64, SMAP is enabled by setting bit 21 of CR4. To access memory in user space on arm64, one either has to clear the PAN bit or use the specialized instructions `ldt*` and `stt*` for loading and storing memory.

On arm64, PAN is only supported on devices based on ARMv8.1 and newer. For older devices there is a software emulated version.

5.4.4 `STRICT_KERNEL_RWX`

`STRICT_KERNEL_RWX` is a configuration option used to make sure that text and ro-data sections are read-only. This option is useful to make it harder for attackers to inject code or perform data-only attacks.

5.4.5 `STRICT_DEVMEM`

Linux provides a special device named `/dev/mem`. This devices allows access to all kernel and user space memory on the system. Switching on the config option

STRICT_DEVMEM will put restrictions on what memory is available through this device. If IO_STRICT_DEVMEM is not enabled, user space can access the PCI space and BIOS code/data regions through /dev/mem. If IO_STRICT_DEVMEM is enabled, only idle io-memory ranges are accessible.

5.4.6 HARDENED_USERCOPY

This option attempts thwart certain types of vulnerabilities by hardening the functions responsible for copying data to/from the kernel. Examples of mitigations include rejecting copy operations that attempt to copy more data to/from a heap allocation than what was allocated. For stack allocations, the kernel checks that the object copied is within the current stack frame.

Most of the hardened usercopy code is located in mm/usercopy.c, with some parts located in heap implementation specific files. When using copy_to_user() and copy_from_user() to copy to and from user space, check_object_size() is called to verify the copy operation. This is a thin wrapper around __check_object_size(), which performs several checks on the object. Figure 5.1 contains the full code listing.

From the code we can see that several properties are checked. First, the code verifies that the address is in fact a valid address and not somethig invalid like a null pointer. It also detects if the object in question (ptr + n) wraps around past the end of memory. If the address belongs to the stack, it is verified that the object belongs to the current stack frame, if possible. If frame checking is not available, the code will check that the address is fully within the process stack. Objects that partially overlaps the stack will be marked as invalid.

5.4.7 Page Table Isolation

Page Table Isolation, or PTI for short, is a countermeasure against attacks like Meltdown [46]. These attacks can leak sensitive kernel space memory from user space on vulnerable CPUs. To mitigate this class of attacks, PTI uses two sets of page tables. One contains a full copy of all memory, including kernel space memory. The other set is meant for user space applications, and only map user space memory and essential kernel memory needed for syscalls and so on. When entering the kernel from user space, the system switches to the page tables mapping all memory, and then switches back again when the kernel returns to user space.

```
/*
 * Validates that the given object is:
 * - not bogus address
 * - fully contained by stack (or stack frame, when available)
 * - fully within SLAB object (or object whitelist area, when available)
 * - not in kernel text
 */
void __check_object_size(const void *ptr, unsigned long n, bool to_user)
{
        if (static_branch_unlikely(&bypass_usercopy_checks))
                return;

        /* Skip all tests if size is zero. */
        if (!n)
                return;

        /* Check for invalid addresses. */
        check_bogus_address((const unsigned long)ptr, n, to_user);

        /* Check for bad stack object. */
        switch (check_stack_object(ptr, n)) {
        case NOT_STACK:
                /* Object is not touching the current process stack. */
                break;
        case GOOD_FRAME:
        case GOOD_STACK:
                /*
                 * Object is either in the correct frame (when it
                 * is possible to check) or just generally on the
                 * process stack (when frame checking not available).
                 */
                return;
        default:
                usercopy_abort("process stack", NULL, to_user, 0, n);
        }

        /* Check for bad heap object. */
        check_heap_object(ptr, n, to_user);

        /* Check for object in kernel to avoid text exposure. */
        check_kernel_text_object((const unsigned long)ptr, n, to_user);
}
EXPORT_SYMBOL(__check_object_size);
```

Figure 5.1: Implementation of `__check_object_size()`

Chapter 6

Forward-edge and backward-edge control-flow integrity in the Linux kernel

6.1 Introduction

This chapter will describe how support for LTO, CFI, and SCS is implemented in the Linux kernel. Before we can use these features, however, the kernel has to be built with clang. In section 6.2, we describe the work that has gone into making this possible. Then in section 6.3 we describe the actual patches implementing LTO, CFI, and SCS in the kernel. In section 6.3.3 we describe how a CFI failure looks like, and how we can fix false positives by patching the kernel. In section 6.4 we describe the SCS implementation in further detail.

6.2 Building the Linux kernel with clang

On newer kernel versions, a lot of patches have been added to both clang and the linux kernel that makes it possible to build a vanilla upstream kernel. The ClangBuiltLinux [18] project has done a significant amount of work to make this possible. They have a huge list of issues regarding compilation, linking, and so on located on their GitHub kernel repository [19]. There is work being done to make it possible to compile the kernel for several of the supported architectures with clang. This could be an interesting topic for further research.

Following is a brief history of some of the different issues that have been resolved to make it possible to compile the Linux kernel with clang.

6.2.1 Variable-length arrays

A *variable-length array* (VLA) is an array whose size is determined at runtime instead of compile time. VLAs are supported by the C99 standard and is usually implemented by allocating room on the stack at runtime for the array. Consider the following C code example:

```
void func(size_t n)
{
        /* VLA */
        char array[n];

        for (size_t i = 0; i < n; i++)
                do_something(array[i]);
}
```

The code is totally useless, but shows how one could use VLAs in C code. The array named `array` has a size determined at runtime by the `n` parameter to `func()`. VLAs have not been very popular in the Linux kernel, and Linus Torvalds has stated his dislike with VLAs on the public kernel mailing list. With the 4.20 kernel, however, VLAs have been completely removed. The main reasons may have been for performance and security, since compilers generated slow code for VLAs and it may lead to security bugs if the array size is not properly checked. Clang does not support VLAs, however, so it was necessary to completely remove them from the kernel to build it with clang.

6.2.2 Assembly goto

The linux kernel uses asm goto statements in its jump label code. An asm goto is a goto statement that allows jumping from assembly to a C label.

Clang versions before 9 did not support this syntax, however, so the kernel had to be built with `JUMP_LABEL` disabled. After version 9, however, asm goto support is added and the kernel can be successfully built.

6.3 LTO, CFI, and SCS in Linux

Support for LTO, CFI, and SCS is merged into the upstream Android kernels and can be used by any vendor by compiling the kernel with clang and enabling a few configuration options. In addition, Sami Tolvanen has a patchset compatible with the upstream kernel [75]. The patchsets for the upstream kernel supports x86_64 as well as arm64. Kees Cook has documented how to boot an x86_64 kernel with CFI enabled on his blog [20].

As of June 6th, 2020, support for SCS on arm64 has been merged into the upstream kernel [1].

The following subsections will describe in further detail how the different features work in the Linux kernel.

6.3.1 Link-time optimization

In this section we will describe the changes that have been made to the Linux kernel to support LTO. We will not describe every single patch, but the most important and interesting changes will be covered here.

For LTO support, almost all of the patches are directed at the build system. First of all some new configuration options are added: `LTO_CLANG` enables LTO, and `THINLTO` enables ThinLTO. The compiler flags used to enable LTO are

[1]commit `533b220f7be4e461a5222a223d169b42856741ef`

`-flto` for full LTO, or `-flto=thin` for ThinLTO. If the clang version supports the flag, `-fsplit-lto-unit` will be used to enable LTO unit splitting which will split bitcode objects into into regular and thin LTO halves, which is enabled for CFI for example. To speed up incremental builds with ThinLTO a cache can be used. The cache directory can be specified with a linker flag [2].

When using loadable modules, it is important that the module is built for the exact same kernel as it is loaded on. Many things can change between kernel versions, and by changing configuration options two kernels with the same version can vary greatly. Structure layouts may change, functions may become deprecated, and new functionality could be added or removed. To make sure that it is safe to load a module, the kernel has an option called `MODVERSIONS`. *modversions* stores checksums in a special section of every module, and when the kernel loads the module it can check if the module is compatible with the running kernel.

LTO produces bitcode files instead of object files, so some tricks have to be used to extract symbols from the bitcode files using the `llvm-nm` utility instead of `objdump` which is normally used.

To reduce binary size, the flags `-mllvm -import-instr-limit=5` are used to limit inlining across translation unit boundaries. The authors report an 11 % decrease in size for a stripped arm64 defconfig vmlinux binary by switching from the default value of 100 to 5.

6.3.2 Control-flow integrity

Like LTO, some additional compiler flags have to be enabled to support CFI. The flag `-fsanitize=cfi` is used to enable CFI. In addition, the kernel is compiled with `-fno-sanitize-cfi-canonical-jump-tables`, which makes the jump tables used by CFI *non-canonical*. First of all, let us discuss what a *canonical* jump table is. By default, clang will replace the address of each function checked by CFI with the address of a jump table entry that will pass CFI checks. This is how clang makes the jump table *canonical*. All the original function symbols get `.cfi` appended to them, and the function symbol will point to the jump table entry which will call the original function. With non-canonical jump tables, the symbol table entry points to the function body instead. A compiler attribute can be used to tell the compiler to use a canonical jump table instead.

Having a canonical jump table can be useful since it allows code not instrumented with CFI checks to take a CFI-valid address of a function. There is, however, increased performance and code size overhead since each exported function has an associated jump table entry even though the function is never used in an indirect call anywhere.

In some cases, however, it is necessary to use canonical jump tables. One example is functions that take the address of a function in assembly code, and later use it in C. When we take the address of a function in C, it will use the jump table entry, thus passing any CFI checks. When the address is taken in assembly, however, we get a reference to the original function body. If this reference is later used to call the function in CFI-checked C code, we will get a CFI violation.

Next, we will discuss the different configuration options added to the kernel

[2] `--thinlto-cache-dir=`

pertaining to CFI. First up, `CFI_CLANG` enables CFI. This option depends on LTO. By default, CFI violations result in a kernel panic. This might not be desirable, especially during development. The option `CFI_PERMISSIVE` can be used to enable permissive CFI mode, where a warning is used instead of a kernel panic when a violation is caught. `CFI_CLANG_SHADOW` speeds up cross-module CFI checks, more on this option later in this section.

All modules now contain a CFI check function. This function is generated by the compiler and is named `__cfi_check()`. The prototype for the function looks like this:

```
extern void __cfi_check(uint64_t id, void *ptr, void *diag);
```

Some parts of the kernel is not instrumented with CFI, for example error handlers, functions that jump to a physical address, and exception callbacks.

Finally we will discuss some of the minor changes that have been done to different parts of the C code. The kernel has something called *kallsyms*, which is used for symbol lookup within the kernel. From user space it can be accessed through the `/proc/kallsyms` file. When both ThinLTO and CFI are enabled, LLVM appends a hash to static function names, which might break tools relying on information from kallsyms. To prevent potential issues, the following code snippet is added to remove hashes from function names:

```
#if defined(CONFIG_CFI_CLANG) && defined(CONFIG_THINLTO)
/*
 * LLVM appends a hash to static function names when ThinLTO and CFI are
 * both enabled, which causes confusion and potentially breaks user space
 * tools, so we will strip the postfix from expanded symbol names.
 */
static inline void cleanup_symbol_name(char *s)
{
        char *res;

        res = strrchr(s, '$');
        if (res)
                *res = '\0';
}
#else
static inline void cleanup_symbol_name(char *s) {}
#endif
```

6.3.3 CFI failure

The following subsection will describe how a CFI failure looks and how it can be fixed in the Linux kernel source code. For the CFI code to be useful on x86_64, it is important that all CFI failures caused by normal kernel code is removed. A CFI kernel that panics because of a wrong function signature is useless in production. When testing, it is fine to use permissive CFI, which simply prints an error message to the kernel log about the failure. For CFI to actually protect against function pointer overwrites, the kernel has to panic when it detects a failure. This is the same behavior as when the kernel detects stack corruption.

When running a kernel with CFI enabled on an Intel NUC a CFI failure was reported in a networking driver. Following is the CFI failure message with some information removed to make it more readable.

```
------------[ cut here ]------------
CFI failure (target: cfg80211_wext_giwname.cfi_jt+0x0/0x10 [cfg80211]):
WARNING: CPU: 0 PID: 2536 at kernel/cfi.c:29
         __ubsan_handle_cfi_check_fail+0x38/0x40
CPU: 0 PID: 2536 Comm: ThreadPoolForeg
       Tainted: G        W         5.5.0-rc7-cfi+ #3
Hardware name:  /NUC6i7KYB, BIOS KYSKLi70.86A.0037.2016.0603.1032
                06/03/2016
RIP: 0010:__ubsan_handle_cfi_check_fail+0x38/0x40
Call Trace:
 __cfi_check_fail+0x1c/0x30 [cfg80211]
 __cfi_check+0x27eb/0x2940 [cfg80211]
 ? 0xffffffffc08ee000
 ? __cfi_slowpath_diag+0x6e/0xe0
 ioctl_standard_call$9fd5dafa701484d9e63973be756c9b3f+0x185/0x1b0
 wext_handle_ioctl+0x15c/0x350
 sock_ioctl$cb487607ef45d635e89da547c2034f0c+0x61/0x470
 ? alloc_empty_file+0x82/0xe0
 ? alloc_file+0x2b/0xf0
 ? stm_char_ioctl$17cda3d452c6c05ccbe20f2bfbcab4d2.cfi_jt+0x8/0x8
 do_vfs_ioctl+0x7ac/0xa60
 ? sock_alloc_file+0xeb/0x140
 __x64_sys_ioctl+0x73/0xa0
 ? __ia32_sys_io_setup.cfi_jt+0x8/0x8
 do_syscall_64+0x7c/0x130
 entry_SYSCALL_64_after_hwframe+0x44/0xa9
---[ end trace 7ef867b86cd5ed42 ]---
```

The kernel was compiled with permissive CFI, which means that it will continue running even though a CFI failure is encountered. When permissive mode is disabled, the kernel will panic instead. The code uses the standard `WARN` macro which will log information like the ID of the current procesor, the PID of the current process, and a stack trace. In addition, the CFI code will print the indirect function call target that caused the failure.

The target that caused this CFI failure was `cfg80211_wext_giwname()`. It was called from `ioctl_standard_call()`. Looking at the prototype of the called function we see that it looks like this:

```
int cfg80211_wext_giwname(struct net_device *dev,
                          struct iw_request_info *info,
                          char *name, char *extra);
```

The next step is to look at the call site to see where the error occurs. See the following code snippet [3]:

[3]defined in `net/wireless/wext-core.c`

```
1  /*
2   * Wrapper to call a standard Wireless Extension handler.
3   * We do various checks and also take care of moving data between
4   * user space and kernel space.
5   */
6  static int ioctl_standard_call(struct net_device *        dev,
7                                 struct iwreq                *iwr,
8                                 unsigned int                cmd,
9                                 struct iw_request_info        *info,
10                                iw_handler                handler)
11 {
12         const struct iw_ioctl_description *        descr;
13         int                                        ret = -EINVAL;
14 
15         /* Get the description of the IOCTL */
16         if (IW_IOCTL_IDX(cmd) >= standard_ioctl_num)
17                 return -EOPNOTSUPP;
18         descr = &(standard_ioctl[IW_IOCTL_IDX(cmd)]);
19 
20         /* Check if we have a pointer to user space data or not */
21         if (descr->header_type != IW_HEADER_TYPE_POINT) {
22 
23                 /* No extra arguments. Trivial to handle */
24                 ret = handler(dev, info, &(iwr->u), NULL);
25 
26                 /* Generate an event to notify listeners of the change */
27                 if ((descr->flags & IW_DESCR_FLAG_EVENT) &&
28                                 ((ret == 0) || (ret == -EIWCOMMIT)))
29                         wireless_send_event(dev, cmd, &(iwr->u), NULL);
30         } else {
31                 ret = ioctl_standard_iw_point(&iwr->u.data, cmd, descr,
32                                 handler, dev, info);
33         }
34 
35         /* Call commit handler if needed and defined */
36         if (ret == -EIWCOMMIT)
37                 ret = call_commit_handler(dev);
38 
39         /* Here, we will generate the appropriate event if needed */
40 
41         return ret;
42 }
```

At line 24 we can see that the handler passed to `ioctl_standard_call()` is called. The handler has the type `iw_handler` which looks like this:

```
typedef int (*iw_handler)(struct net_device *dev,
                          struct iw_request_info *info,
                          union iwreq_data *wrqu,
                          char *extra);
```

Comparing iw_handler to the declaration of cfg80211_wext_giwname() we see that the type of the third parameter does not match. The handler expects an argument of type union iwreq_data while the function actually takes a char *. The fix here is to make sure that the two function signatures match. To make these match we can change cfg80211_wext_giwname() to use the expected parameter type instead of char *. Following is an excerpt of the definition of iwreq_data with comments removed for brevity:

```
union iwreq_data {
        char             name[IFNAMSIZ];

        struct iw_point essid;
        struct iw_param nwid;
        struct iw_freq  freq;
        /* [... ] */
};
```

Also see the following code snippet with the definition of cfg80211_wext_giwname():

```
int cfg80211_wext_giwname(struct net_device *dev,
                          struct iw_request_info *info,
                          char *name, char *extra)
{
        strcpy(name, "IEEE 802.11");
        return 0;
}
EXPORT_WEXT_HANDLER(cfg80211_wext_giwname);
```

iwreq_data is a union type, so simply casting it to a char * works fine here. However, to make it compatible with the iw_handler type we have to change the char * parameter to a iwreq_data and copy the name into the name field of this union instead. A patch to fix this issue can be found in figure 6.1. Most of the CFI failures can be, and have been, fixed this way. Fixing these simple CFI failures should not make huge changes to the original code.

6.4 ShadowCallStack

In section 4.6.3 we briefly introduced ShadowCallStack, or SCS for short. In this section we will dive deeper into how SCS is implemented in the Linux kernel. As of early April, 2020, the SCS patchset [4] consists of 12 commits with 31 changed files. In total there is 440 additions and 7 deletions. Compared with the patchsets for LTO and CFI this is a little simpler.

Let us start with the changes to the build system. A new configuration option, ARCH_SUPPORTS_SHADOW_CALL_STACK is added so that architectures can signal that they support SCS. To enable SCS, the SHADOW_CALL_STACK option is used. The compiler also needs to support SCS, which is currently only clang versions larger than or equal to 7.0. As the official documentation for SCS states [48], the compiler flag -ffixed-x18 has to be enabled on arm64 since the SCS implementation uses the x18 register to store the shadow stack pointer. To

[4]from https://github.com/samitolvanen/linux

```
---
 include/net/cfg80211-wext.h | 2 +-
 net/wireless/wext-compat.c  | 4 ++--
 2 files changed, 3 insertions(+), 3 deletions(-)

diff --git a/include/net/cfg80211-wext.h b/include/net/cfg80211-wext.h
index ad77caf2ffde..f634a6c8f6a9 100644
--- a/include/net/cfg80211-wext.h
+++ b/include/net/cfg80211-wext.h
@@ -19,7 +19,7 @@
  */
 int cfg80211_wext_giwname(struct net_device *dev,
                           struct iw_request_info *info,
-                          char *name, char *extra);
+                          union iwreq_data *wrqu, char *extra);
 int cfg80211_wext_siwmode(struct net_device *dev, struct iw_request_info *info,
                           u32 *mode, char *extra);
 int cfg80211_wext_giwmode(struct net_device *dev, struct iw_request_info *info,
diff --git a/net/wireless/wext-compat.c b/net/wireless/wext-compat.c
index cac9e28d852b..1b773ac87b09 100644
--- a/net/wireless/wext-compat.c
+++ b/net/wireless/wext-compat.c
@@ -25,9 +25,9 @@

 int cfg80211_wext_giwname(struct net_device *dev,
                           struct iw_request_info *info,
-                          char *name, char *extra)
+                          union iwreq_data *wrqu, char *extra)
 {
-        strcpy(name, "IEEE 802.11");
+        strcpy(wrqu->name, "IEEE 802.11");
         return 0;
 }
 EXPORT_WEXT_HANDLER(cfg80211_wext_giwname);
--
2.17.1
```

Figure 6.1: cfg80211_wext_giwname() CFI patch

```
/*
 * Called when gcc's -fstack-protector feature is used, and
 * gcc detects corruption of the on-stack canary value
 */
__visible void __stack_chk_fail(void)
{
        panic("stack-protector: Kernel stack is corrupted in: %pB",
                __builtin_return_address(0));
}
EXPORT_SYMBOL(__stack_chk_fail);
```

Figure 6.2: `__stack_chk_fail()` in the Linux kernel

```
static inline void scs_overflow_check(struct task_struct *tsk)
{
        if (unlikely(scs_corrupted(tsk)))
                panic("corrupted shadow stack detected inside scheduler\n");
}
```

Figure 6.3: Shadow stack overflow handling in the Linux kernel

enable SCS, the flag `-fsanitize=shadow-call-stack` is used. SCS can be disabled for certain functions by using the `no_sanitize("shadow-call-stack")` attribute [5]. This is attribute is defined in the kernel sources as `__noscs` and is used to turn off SCS in EFI and hypervisor code, for example.

Most of the SCS magic is done by the compiler, but most of the in-kernel implementation is done in `kernel/scs.c`. Shadow stacks are allocated on the heap using a kmalloc cache named `scs_cache` with a size of 1 KiB. The flags used when allocating a new stack are `(GFP_KERNEL | __GFP_ZERO)`. `GFP_KERNEL` is typically used for allocations that do not have special requirements like not being able to sleep. `__GFP_ZERO` is basically the kernel equivalent of allocating memory with `calloc()`, which means that the memory is zeroed. This ensures that there is no stale data in the shadow stack.

To check for stack corruption, a simple check resembling the stack canary checks described in section 3.5.2 is used. Instead of a random value, an illegal pointer value is used to mark the end of the shadow stack. If this value is overwritten, the stack has been corrupted and the kernel will panic. This behavior is similar to what happens when the Linux kernel detects a stack overflow. When the stack protector is enabled, function epilogues are instrumented with code that checks the stack cookie (more details in section 3.5.2). If corruption is detected `__stack_chk_fail()` is called to handle the error. In figure 6.3 we can see the implementation for handling shadow stack overflow, and in figure 6.2 we see the implementation for handling normal stack corruption.

The shadow stack pointer, *ssp*, was first stored as a base/offset pair in the `thread_info` struct related to each task on the system. Recently this was switched to storing the absolute value of the `ssp` for performance reasons. This is how the modified `thread_info` struct looks like after applying the SCS patches:

[5] `__attribute__((no_sanitize("shadow-call-stack")))` on a function declaration

```
struct thread_info {
        unsigned long                   flags;
        mm_segment_t                   addr_limit;
#ifdef CONFIG_ARM64_SW_TTBR0_PAN
        u64                            ttbr0;
#endif
        union {
                u64                    preempt_count;
                struct {
#ifdef CONFIG_CPU_BIG_ENDIAN
                        u32            need_resched;
                        u32            count;
#else
                        u32            count;
                        u32            need_resched;
#endif
                } preempt;
        };
#ifdef CONFIG_SHADOW_CALL_STACK
        void                           *shadow_call_stack;
#endif
};
```

The only field added is the `shadow_call_stack` pointer at the end of the struct. Some minor changes have been done to the core arm64 code to support SCS. Upon entry to the kernel from user space, the shadow stack has to be loaded from the `task_struct` into `x18`. Conversely, when returning to user space from the kernel the shadow stack pointer is saved. These operations are perfomed with the `scs_load` and `scs_save` assembly code macros, displayed below:

```
.macro scs_load tsk, tmp
ldr        x18, [\tsk, #TSK_TI_SCS_SP]
.endm

.macro scs_save tsk, tmp
str        x18, [\tsk, #TSK_TI_SCS_SP]
.endm
```

TSK_TI_SCS_SP is the offset of the `shadow_call_stack` member in the `thread_info` struct.

Chapter 7

Performance benchmarks

7.1 Introduction

In this chapter we will discuss chosen performance benchmarks, including what hardware and software was chosen, why the benchmark was chosen, and how to perform the benchmark.

The goal is to measure the performance impact on the kernel with LTO, CFI, and SCS. To have a baseline to compare against, kernels compiled with gcc and clang without any special optimizations will be used. Since several companies and projects are starting to adopt clang/LLVM as their toolchain of choice, it would be interesting to see if there are any notable differences in performance between gcc and clang. The kernel configurations in table 7.1 will be benchmarked on x86_64. For arm64, the configurations in table 7.2 will be benchmarked.

compiler	LTO	inline limit	CFI
gcc	no	N/A	no
clang	no	N/A	no
clang	yes	5	no
clang	yes	100	no
clang	yes	5	yes
clang	yes	100	yes

Table 7.1: x86_64 kernel configurations

In the patchset for LTO support in the kernel, the inline limit is set to 5, instead of the default of 100. This decreases the size of the resulting kernel binary at the cost of (potential) performance gains. As the patchset is meant for Android phones, where size is of greater concern than on laptops and servers, for example, it makes sense to limit inlining. Since binary size is less important on x86_64, we will increase the limit to see how much impact it has on performance and size. Although inlining should increase performance, the increased size may also negatively impact performance if it leads to worse cache locality or other similar issues. The patches implementing a configurable limit can be found in section B.2. This limit is controlled with a linker flag called

compiler	LTO	inline limit	CFI	SCS
gcc	no	N/A	no	no
clang	no	N/A	no	no
clang	yes	5	no	no
clang	yes	100	no	no
clang	yes	5	yes	no
clang	yes	100	yes	no
clang	yes	5	yes	yes
clang	yes	100	yes	yes

Table 7.2: arm64 kernel configurations

`-import-instr-limit=N`. The number, N, used controls the maximum number of instructions for inlined functions. The limit originally used, 5, means that only functions with less than 5 instructions are inlined [1].

Since our benchmarks have to measure the performance of the whole kernel, we employ similar benchmarks to other papers dealing with kernel performance [58] [63]. More on previous work in section 7.2.

Performance benchmarks for x86_64 are performed on real hardware and with QEMU/KVM [11]. The hardware used is a Dell XPS laptop with a 1.80GHz Intel Core i7-8550U CPU, 16 GB RAM, and 512 GB nvme drive. The distro used is Ubuntu 18.04 LTS. To make the benchmarks more reliable, Intel SpeedStep and Intel TurboBoost is disabled from the BIOS. This ensures that the CPU cannot scale up and down as it pleases, which could affect the results. The QEMU version used is the newest in the Ubuntu 18.04 package repositories as of January 2020, which is version `2.11.1` [2]. The image used for QEMU is the same as the one used in the syzkaller project [71]. syzkaller is a an "unsupervised coverage-guided kernel fuzzer" that has found numerous bugs in the Linux kernel and other kernels. The VM image used is based on Debian and the project provides a script to create the image for convenience [3].

When running the x86_64 benchmarks on the Dell XPS, the computer was booted and the desktop environment stopped to minimize the amount of noise from other programs running simultaneously with our benchmark. See section A.5 for the full benchmarking script. First, gdm (default login manager on Ubuntu) is stopped. Then, ssh, redis, nginx, and apache2 is stopped. redis, nginx, and apache2 is later started when they are needed. LMBench is the first benchmark up, followed by redis. Then nginx and apache2 is benchmarked using ApacheBench. Finally, throughput benchmarks using the WireGuard protocol will be performed. More information on WireGuard can be found in section 7.6. The Python scripts used for visualizing the results can be found in section A.6.

The HiKey 960 is considered a reference board for Android, which means that it is supported by the Android Open Source Project (AOSP). The HiKey has a 4 Cortex A73, 4 Cortex A53 Big.Little CPU, ARM Mali G71 Mp8 GPU, 4GB LPDDR4 DRAM, and 32GB UFS flash storage. This board will be used to perform the arm64 benchmarks.

[1]See the LLVM sources: `llvm/lib/Transforms/IPO/FunctionImport.cpp` in the function `ImportInstrLimit()`.

[2]Debian 1:2.11+dfsg-1ubuntu7.21

[3]see `tools/create-image.sh` in the syzkaller sources

For Android, it does not really make sense to run benchmarks for web servers like nginx and apache2. We could have performed the same benchmarks if we targeted servers running arm64, for example. Redis is also more commonly used on servers and desktop computers. WireGuard has not made it into the Android common kernel, so we will skip that as well.

So what should we benchmark on Android? As discussed in section 2.3, binder is a popular IPC mechanism on Android. Binder is implemented in the kernel, making it a perfect component to benchmark since it is heavily used by apps and it will be directly affected by LTO, CFI, and SCS. The benchmarks chosen are based on the official Android documentation for performance testing [5]. Throughput and latency tests for binder and hwbinder are included in the AOSP. To make sure that CPU throttling does not affect the benchmarks it has been turned off by setting the Linux CPU scaling governor to `performance`. This mode makes sure that the CPU runs at maximum frequency. The script used for running the benchmarks can be found in the appendix A.5.

For Android the `hikey-linaro-android-4.19` kernel was chosen. Kernel versions `hikey-linaro-android-4.14`, `hikey-linaro-android-4.19`, and `common-android-5.4` are supported on the HiKey 960. At the time of compilation, the newest commit for the 4.19 kernel was `03a6248` [4]. For x86_64 kernel version `5.7.0-rc2` was used.

7.2 Motivation and previous work

This section will present the motivation for documenting the performance impact of LTO, CFI, and SCS. We will refer to previous work that is relevant for this thesis. However, the only previous research found on LTO, CFI, and SCS performance was done by Google [74] [76]. Since we did not find any research pertaining to clang's LTO, CFI, and SCS in the Linux kernel, previous work that benchmarks the Linux kernel have been used. The chosen research have provided several useful techniques and software to perform benchmarks on Linux.

The reason for choosing both hardware and virtualized benchmarking platforms for x86_64 is because of the widespread use of virtualization on x86_64. Examples of such usage are cloud providers such as Amazon, Google, and Digital Ocean. It might be interesting for cloud providers to see if enabling LTO can improve the performance of their virtual machines and if the performance hit is negligible, if it is worth it to increase security by enabling CFI. The arm64 benchmark is focused on real hardware, since most arm64 devices are smartphones. Knowing the tradeoffs between security and performance by using LTO/CFI can be beneficial for smartphone vendors or other manufacturers of arm64 devices.

When Google first wrote a blog post on CFI, they wrote the following about the performance impact: "CFI checks naturally add some overhead to indirect branches, but due to more aggressive optimizations, our tests show that the impact is minimal, and overall system performance even improved 1-2% in many cases." [74] These are very interesting results indeed, but no further explanation of how the benchmarks were performed was given. Thus it is interesting to see if the performance impact actually is minimal and to provide additional

[4] `03a6248cae932550d4d45eb511eb25b87aef0c1c`

benchmarks like binary size and compilation time. In the following sections we will briefly discuss some of the benchmarks and why we perform them.

The following sections will look at some papers that have inspired the performance benchmarking for this thesis. However, no previous research on the performance impact of clang, LTO, and CFI on x86_64 was found.

kMVX: Detecting Kernel Information Leaks with Multi-variant Execution

Österlund et al. [63] performed several benchmarks of their kernel security feature kMVX. The project runs multiple diversified kernel variants simultaneously to protect against information leak vulnerabilities. They perform benchmarks using both micro and macro benchmarks. The micro benchmarks used are LMBench and stress-ng. For macro benchmarks they use ApacheBench to test web server performance, redis-benchmark, and pbzip2.

Open-source virtualization. Functionality and performance of Qemu/KVM, Xen, Libvirt and VirtualBox

In his Master's Thesis, Jan Magnus Granberg Opsahl [62] compares the performance of different virtualization software. Namely QEMU, Xen, Libvirt, and VirtualBox. We will only be using QEMU for the performance benchmarks in this thesis, but some of the software used for benchmarking is still interesting for our purposes. He employs the following benchmarking suites:

- Cachebench
- LMBench
- Linpack
- IOZone

Just like the paper from Österlund et al. [63], he uses LMBench. The other tools are not used in that research, however. Cachebench is a part of the LLCbench benchmarking suite and is a tool used to determine some parameters about an architectures memory subsystem. The goal of the benchmark is to parameterize the performance of multiple levels of caches present in the processor. This benchmark is great for measuring cache performance, but is not very relevant to this thesis since the CPU caches should not be too affected by the running kernel while benchmarking.

Another benchmarking suite used is Linpack, which "measures the peak value of floating point computational power, also known as GFLOPS" [62]. While this benchmark is great for testing the performance of virtual machines or to test hardware directly, this is another benchmark where the running kernel should not have any impact.

Lastly, IOZone is used. This is a file system benchmark that generates and measures a variety of file operations. According to the manual, the benchmark tests file I/O performance using the following operations: read, write, reread, re-write, read backwards, read strided, fread, fwrite, random read/write, pread/pwrite variants.

DROP THE ROP. Fine-grained Control-flow Integrity for the Linux Kernel

In this paper, Moreira et. al [58] presents a kernel-level fine-grained CFI mechanisms and measures its performance called kCFI. They also present a novel technique called Call Graph Detaching (CGD) which enables the construction of more precise CFGs. When evaluating the performance of kCFI they use LMbench for micro benchmarking just like a lot of the other previously mentioned papers do. For macro benchmarks they use the following tests from the Phoronix Test Suite [56]: IOZone, Linux Kernel Unpacking, PostMark, Timed Linux Kernel Compilation, GnuPG, Openssl, PyBench, Apache Benchmark, PHPBench, Dbench, and PostgreSQL. They ran a lot more macro benchmarks than done in this thesis, but the Apache Bencmark was the one where they saw the biggest performance difference. They measured a 2% increase in code size when applying kCFI and a 4% increase when they applied kCFI + CGD.

OpenMandriva

OpenMandriva Lx [60] is one Linux distribution where packages are built using clang with LTO enabled [61]. They have switched to using clang as their default compiler toolchain instead of gcc, which is the norm for most Linux distributions. In addition they have started building their Linux kernels with clang as well. LTO is not supported for the kernel yet.

7.3 Binary size

For servers, laptops, smartphones, etc. storage might not be an issue anymore. But for certain devices with very limited resources, like some IoT devices, binary size may have a huge impact. Thus we would like to see if enabling LTO and CFI greatly increases the binary size, and if so, if there are ways we can alleviate this issue for devices with strict resource requirements.

7.4 Compile time

LTO is a complex extra step that is taken in the compilation process. For huge projects like the Linux kernel there are a lot of object files that have to be analyzed. Adding CFI into the mix will probably increase the compilation time as well. Thus we would like to see if the compiliation time is greatly impacted by enabling these compiler optimizations. If the impact is huge it might be desirable for kernel developers to disable them during development, and only turning them back on when code is ready to ship. There are tools that can improve compilation times like `ccache` which caches compilation of object files, which can then be reused unless the source files for that object file have changed. `ccache` does not alleviate the time spent during LTO, however.

The Dell XPS laptop has been used to perform most of the benchmarks, but there were some issues when compiling the kernel with full LTO on this hardware. Since full LTO requires a lot of RAM, it is not possible to compile more than a very basic kernel configuration with full LTO. The linking step requires more than the 16 GB of RAM available on the laptop. Because of these issues, full LTO was not included.

7.5 Kernel performance

Based on previous work done with Linux kernel performance, the following micro and macro benchmarks were chosen for this thesis. These are performed in addition to binary size and compilation time.

name	type
LMBench	micro
redis-benchmark	macro
ApacheBench	macro
WireGuard bandwidth	macro

Table 7.3: Micro and macro benchmarks

LMBench has a lot of different benchmarks which are mainly divided into two categories: latency and bandwidth. See table 7.4 for a description of the different benchmarks.

name	lmbench tool	description
read bandwidth	bw_file_rd io_only	read a file in 64KB blocks
read open2close bandwidth	bw_file_rd open2close	read a file in 64KB blocks, with open/close
Mmap read bandwidth	bw_mmap_rd mmap_only	map file in memory and read
Mmap read open2close bandwidth	bw_mmap_rd open2close	map file in memory and read with open/close
libc bcopy unaligned	bw_mem bcopy	measures how fast the system can bcopy data
libc bcopy aligned	bw_mem bcopy conflict	measures how fast the system can bcopy data
Memory bzero bandwidth	bw_mem bzero	measures how fast the system can bzero memory
unrolled bcopy unaligned	bw_mem fcp	measures the time to copy data from one location to another
unrolled partial bcopy unaligned	bw_mem cp	measures the time to copy data from one location to another
Memory read bandwidth	bw_mem frd	measures the time to read data into the processor
Memory partial read bandwidth	bw_mem rd	measures the time to read data into the processor
Memory write bandwidth	bw_mem fwr	measures the time to write data to memory
Memory partial write bandwidth	bw_mem wr	measures the time to write data to memory
Memory partial read/write bandwidth	bw_mem rdwr	measures time to read data into memory and then write data to the same memory location

Table 7.4: LMBench bandwidth benchmarks description

7.6 WireGuard

In addition to the previously mentioned benchmarks, another interesting benchmark was added to this thesis. In version 5.6 of the Linux kernel, support for WireGuard was added. WireGuard [26] is a secure network tunnel which aims to replace IPsec and other solutions like OpenVPN. WireGuard is much less complex than IPsec and uses modern cryptographic primitivese All the core functionality is implemented directly in the Linux kernel. Since WireGuard is a part of the Linux kernel, it would be interesting to see if LTO and CFI have any effect on its performance. The WireGuard source code contains a script for running tests, but we will be using a slightly modified version of that script used in Phoronix Test Suite [5].

The script used to test WireGuard is located at `tools/testing/selftests/wireguard/netns.sh` in the Linux kernel sources.

[5]see A.7 for more information

Both of the scripts start by creating a network topology consisting of three network namespaces. The following diagram shows the topology [6]:

```
 --------------------     -----------------------------     --------------------
|   $ns1 namespace   |   |       $ns0 namespace        |   |   $ns2 namespace   |
|                    |   |                             |   |                    |
| ________           |   |          ________           |   |            _______ |
||  wg0   |__________|___|_________|   lo   |__________|___|___________| wg0   ||
||________|_________ |   |  _______|________|________  |   | __________|_______||
||192.168.241.1/24  ||   | |(ns1)         (ns2)      | |   ||192.168.241.2/24  ||
||fd00::1/24        ||   | |127.0.0.1:1   127.0.0.1:2| |   ||fd00::2/24        ||
||__________________||   | |[::]:1        [::]:2     | |   ||__________________||
|____________________|   | |_________________________| |   |____________________|
                         |_____________________________|
```

Namespaces [38] [39] [40] [41] [42] [43] [27] are heavily used in container technology like Docker [35]. They allow wrapping a global resource in an abstraction that makes the processes within that namespace believe that they have their own instance of that global resource. Linux provides many different namespaces, including: cgroup, IPC, network, mount, and PID.

The WireGuard test script creates three network namespaces named `$ns0`, `$ns1`, and `$ns2` in the diagram above. `$ns1` and `$ns2` do not talk directly to each other, but talk through `$ns0`. All the traffic goes through the loopback interface in `$ns0`. This is a great benchmark for WireGuard, as it puts a lot of pressure on the underlying cryptographic protocols used to protect the communication between two WireGuard peers.

[6] taken from `netns.sh`

Chapter 8

Results

8.1 Introduction

In the following sections we will discuss the results from benchmarks performed. First, we will summarize the configurations used. The first and most common configuration is a kernel compiled using gcc version `9.3.0-10` with a fairly new standard Linux kernel configuration from Ubuntu (`5.3.0-26-generic`). Then we have a kernel compiled with clang 11 [1]. The configuration is the same for gcc and clang. Next up we have two kernels compiled with clang and LTO enabled. The first one has the inlining limit set to 5, while the other one (LTOv2) has it set to 100. Finally, we have two kernels with CFI enabled. As with the LTO kernels, one has an inline limit of 5, while the other (CFIv2) has it set to 100. Note that LTOv2 may be referred to as LTO100, and CFIv2 referred to as CFI100 in the benchmark results.

In section 8.3 we will look at the binary sizes of the different compiled kernels. Then in section 8.4 we will look at compilation time. We will then move on to micro benchmarks with LMBench in section 8.5.1. Finally, we will look at different macro benchmarks. In section 8.6.1 we will look at redis performance, followed by nginx/apache2 performance measured using ApacheBench in section 8.6.2. The last macro benchmark is for WireGuard and can be found in section 8.6.3.

After looking at the benchmark results we will summarize our findings in section 8.8 and finally discuss future work in section 8.9.

8.2 QEMU results

The x86_64 benchmarks were performed in a QEMU virtual machine in addition to running them on real hardware. Unfortunately these results were not accurate enough to use, and it was not possible to get a good comparison. The benchmarks were performed in the same way as on the Dell XPS. The VM was booted and the benchmarking script was run after background processes had been stopped.

The results varied too much between each run, and there was not enough time to figure out why the results were so unstable.

[1] `11.0.0-++20200414100631+5c1d1a62e37-1 exp1 20200414201304.3176`

8.3 Binary size

In table 8.1 we can see the resulting sizes of the different kernels. Note that all sizes are displayed in bytes. gcc produces the smallest kernel, followed by clang, then the two LTO kernels, and finally kernels with LTO and CFI. CFI produces the largest kernel, which is expected because of all the extra validation code and jump tables. The percentage increase in tabe 8.1 is measured from the smallest size. The CFI kernel is 20.21 % larger in size than the kernel compiled with gcc. We can see that the increased inlining greatly increases binary size. There is an 11.67 % increase in size from LTO to LTOv2, and a 10.66 % increase from CFI to CFIv2.

kernel version	compiler/feature	size	percent increase
5.7.0-rc2	gcc	8409920	N/A
5.7.0-rc2	clang	8737856	3.90 %
5.7.0-rc2	LTO	8810432	4.76 %
5.7.0-rc2	LTOv2	9791520	16.43 %
5.7.0-rc2	CFI	10109952	20.21 %
5.7.0-rc2	CFIv2	11005760	30.87 %

Table 8.1: Linux kernel binary sizes

In table 8.2 we can see the size of the Android kernel images. The size increases are relatively similar to the x86_64 results. From LTO to LTOv2 there is an 13.20 % increase in size. From CFI to CFIv2 there is a 10.12 % increase. Enabling SCS increases the size slightly, but it is less than 1 %.

kernel version	compiler/feature	size	percent increase
hikey-linaro-android-4.19	clang	9984787	N/A %
hikey-linaro-android-4.19	LTO	10038070	0.53 %
hikey-linaro-android-4.19	LTOv2	11355252	13.73 %
hikey-linaro-android-4.19	CFI	11398476	14.16 %
hikey-linaro-android-4.19	SCS	11433115	14.51 %
hikey-linaro-android-4.19	CFIv2	12409270	24.28 %
hikey-linaro-android-4.19	SCSv2	12446793	24.66 %

Table 8.2: Android Linux kernel binary sizes

8.4 Compilation time

To measure the compilation time for the different kernel configurations the scripts in section A.2 was used. The script measures resource usage for the compilation process using the `time` utility. `time` is commonly found on Linux systems, and outputs the execution time for a program by default. On arm64, a local build configuration (see section A.3) was used when building, and `time` was used to track the elapsed time. On arm64, the full script is not included in section A.2, but the `time` utility was used there as well. Due to time constraints, the results for x86_64 are not included.

In the following snippet we run the `find` command to locate all files named `vmlinux` in our home folder. To measure the time elapsed we simply put `time` in front of the command.

```
$ time find ~ -name vmlinux &> /dev/null
find ~ -name vmlinux &> /dev/null
7,86s user 10,71s system 52% cpu 35,224 total
```

In the previous example, the `find` command spent a total of 35.224 seconds.

kernel version	config	compilation time	ms	increase
hikey-linaro-android-4.19	clang	14m 17s 44ms	857044	N/A %
hikey-linaro-android-4.19	LTO	17m 10s 84ms	1030084	20.19 %
hikey-linaro-android-4.19	SCS	18m 27s 03ms	1107003	29.17 %
hikey-linaro-android-4.19	CFI	18m 40s 87ms	1120087	30.69 %
hikey-linaro-android-4.19	LTOv2	21m 00s 82ms	1260082	47.03 %
hikey-linaro-android-4.19	SCSv2	21m 56s 60ms	1316060	53.56 %
hikey-linaro-android-4.19	CFIv2	22m 19s 60ms	1339060	56.24 %

Table 8.3: Android kernel compile times

`ccache` was not used for any of these benchmarks, and all of the kernels were built using new output directories so that no old object files could be reused. Everything was built from scratch. `ccache` can greatly speed up the build process after the kernel has been built once, since it will cache object files created during compilation that can be reused later. `ccache` will not help speed up the LTO and linking steps, however. Clang does have support for a ThinLTO cache which can help speed up incremental compilation.

8.5 Micro benchmarks

In the following section we will describe the results from the LMBench micro benchmarks performed on x86_64.

8.5.1 LMBench

To make the results from LMBench more readable, they have been split into categories based on their execution time. Short benchmarks are grouped together, and longer ones are grouped together. In figure 8.1, we see the results for the simple syscall benchmarks, signal handlers, protection fault, pipe latency, and `AF_UNIX` socket stream latency. In figure 8.2, we see the results for the different select syscall benchmarks. We can see process benchmarks in figure 8.3. The bandwidth benchmarks can be found in figures 8.4, 8.5, 8.6, 8.7, 8.8, 8.9, 8.10, 8.11, 8.12, and 8.13.

For the simple LMBench benchmarks the results are pretty even between the different kernel configurations. The simple syscall benchmark was very unstable between different runs of the benchmarks, and we can thus ignore it. The gcc kernel was the fastest one in 7 out of 11 of the simple benchmarks. The CFI kernels performed worst on average.

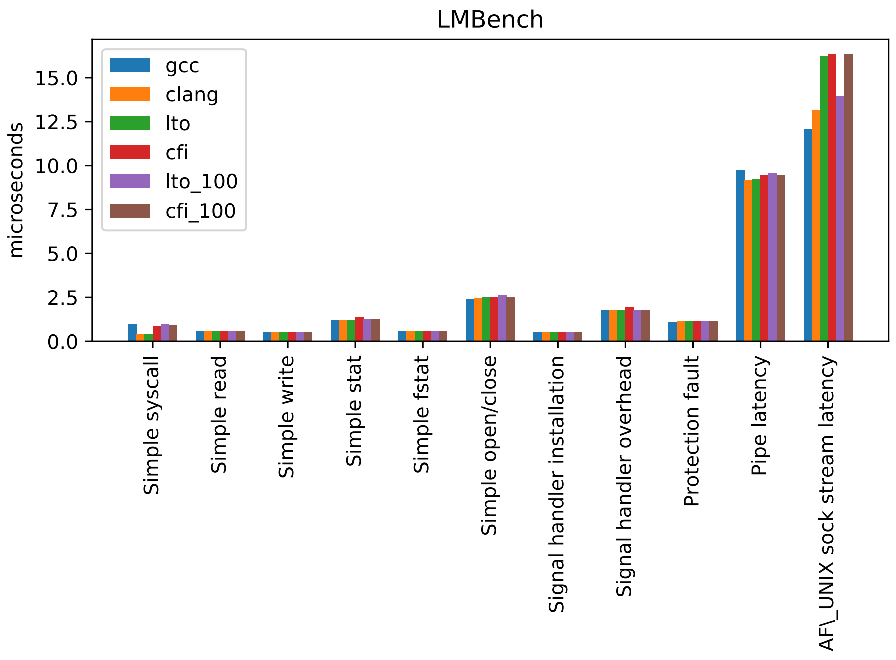


Figure 8.1: Simple LMBench results

For the `select()` syscall results in figure 8.2, gcc was the fastest kernel in every single benchmark. The CFI kernels performed worst on average here as well.

In the process benchmarks, where the timing of process creation is measured, the LTOv2 kernel was the fastest in all of the benchmarks.

It is not very surprising that the CFI kernels performed worst, as CFI adds a lot of overhead to every single indirect function call. There was not a very big difference in the performance of the different kernels, however, which may indicate that CFI does not slow down the kernel that much in these micro benchmarks.

To get a better understanding of why the benchmarks might differ between the kernel configurations, we will examine a syscall in the kernel images. We will start by examining the `stat` syscall, which is implemented in `fs/stat.c`. The size of the structure passed to the `stat` syscall has changed over time, leading to several versions of the syscall being present. We will focus on the newest version, which is `newstat`. In figure 8.14 we see the definition of `newstat` in the kernel sources.

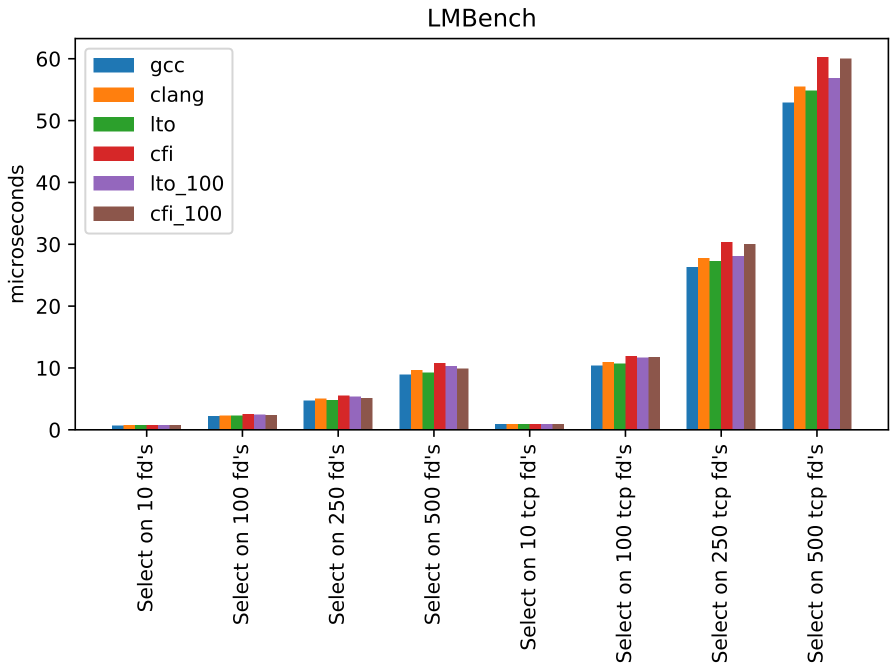


Figure 8.2: LMBench select results

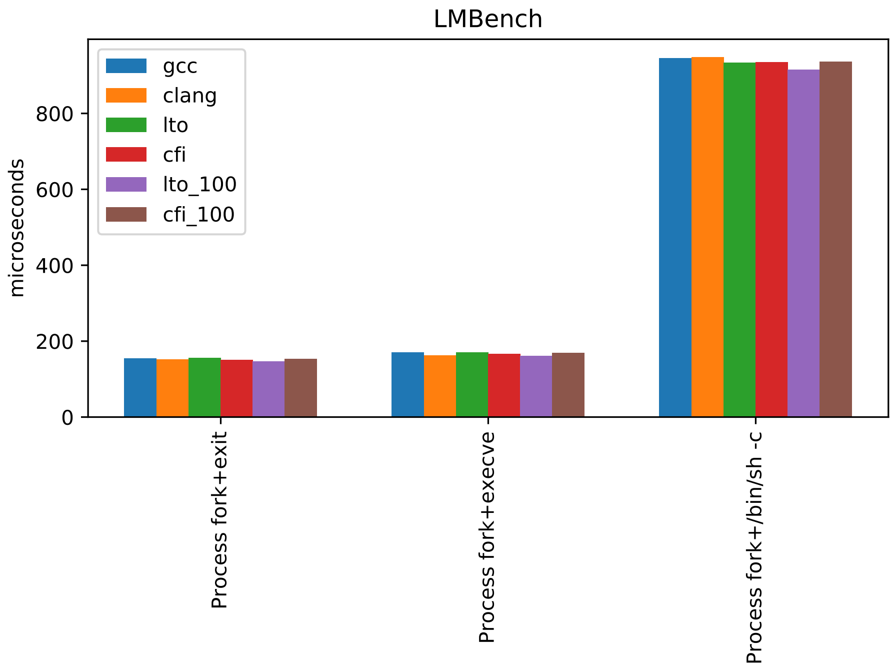


Figure 8.3: LMBench process results

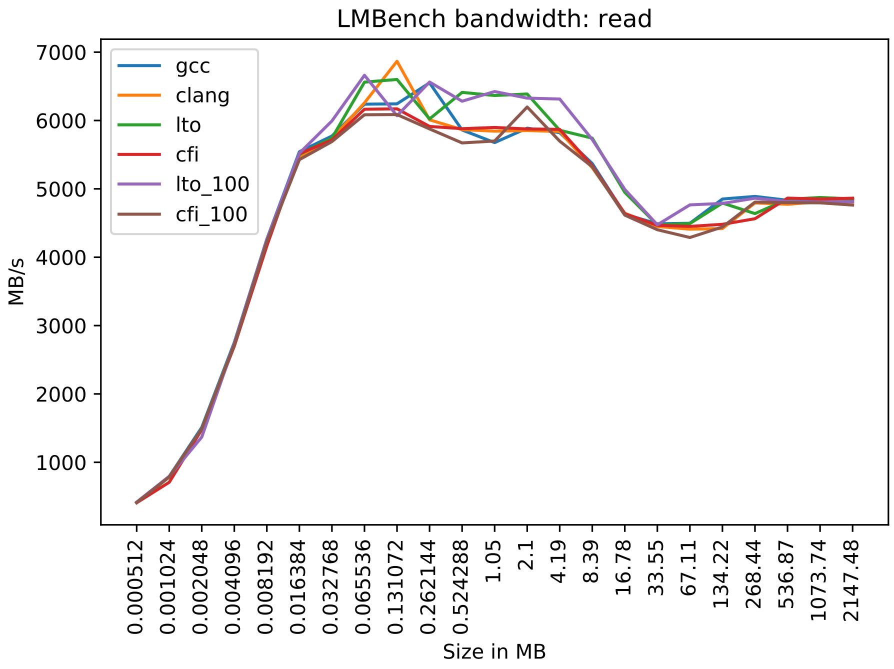


Figure 8.4: LMBench bandwidth read results

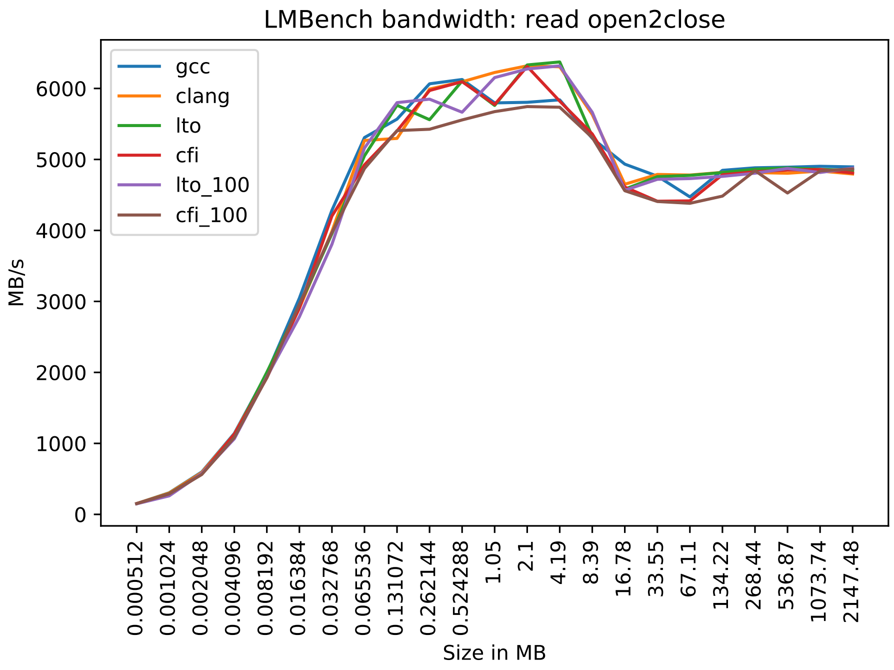


Figure 8.5: LMBench bandwidth read open2close results

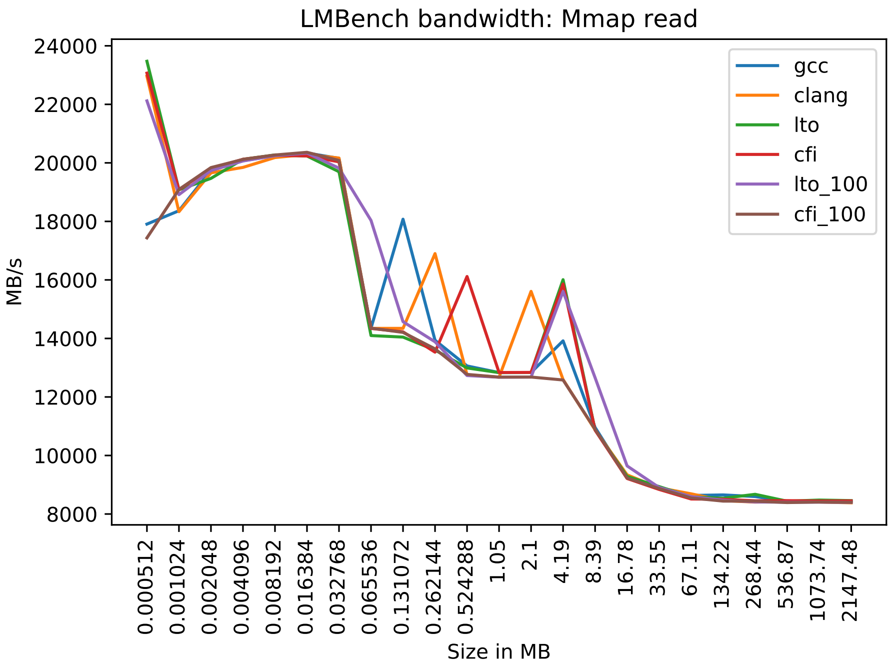


Figure 8.6: LMBench bandwidth mmap read results

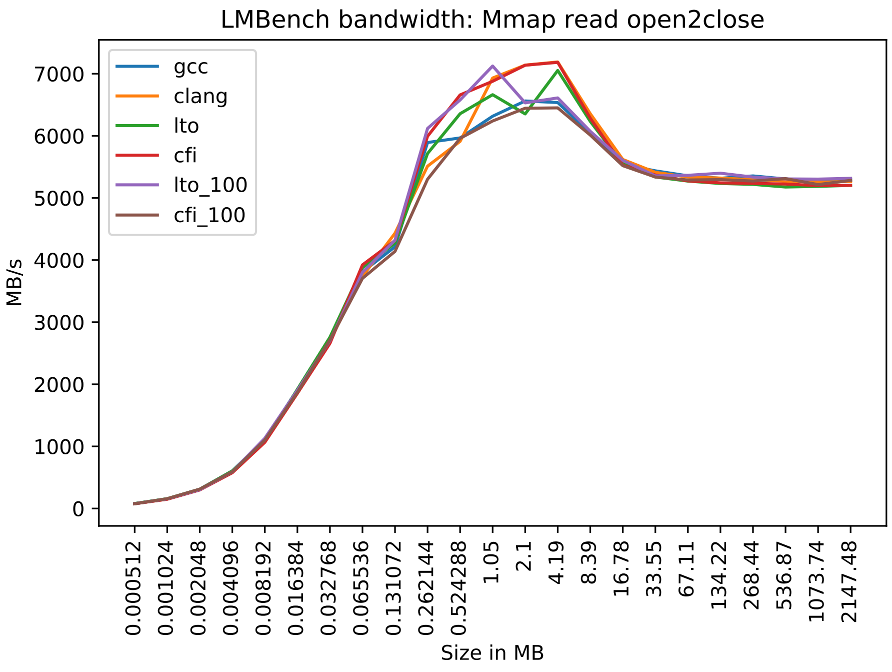


Figure 8.7: LMBench bandwidth mmap read open2close results

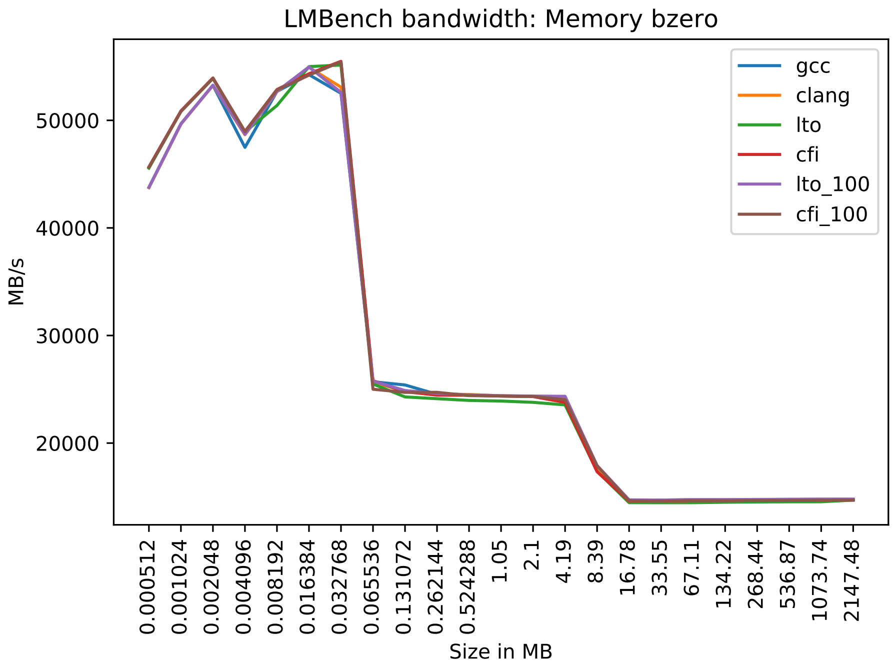


Figure 8.8: LMBench bandwidth bzero results

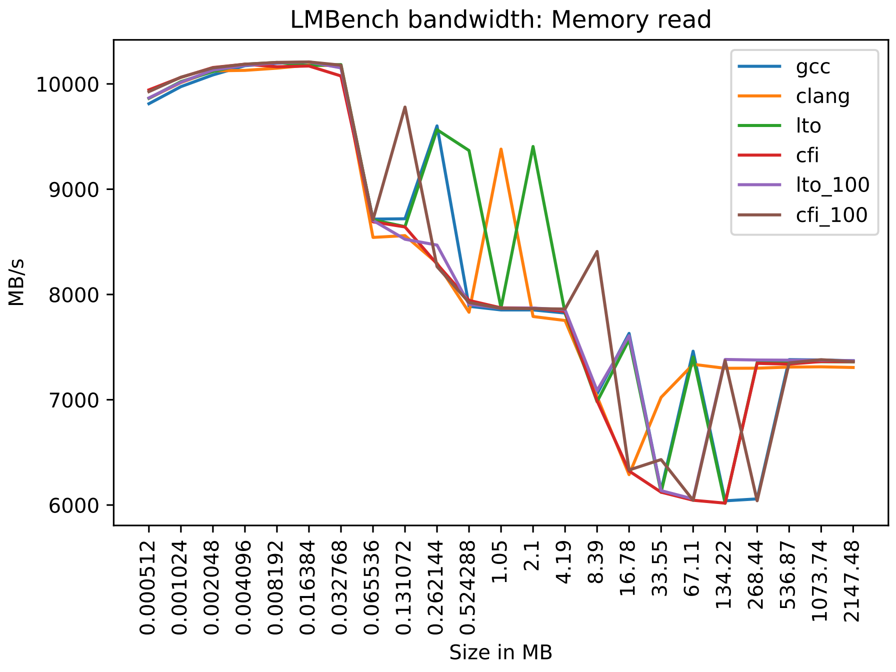


Figure 8.9: LMBench bandwidth memory read results

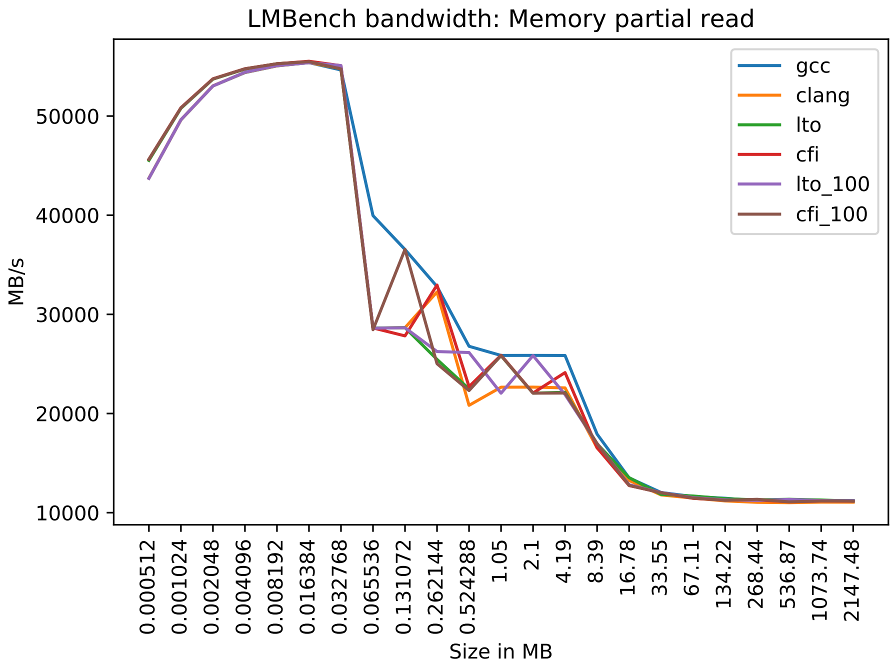


Figure 8.10: LMBench bandwidth memory partial read results

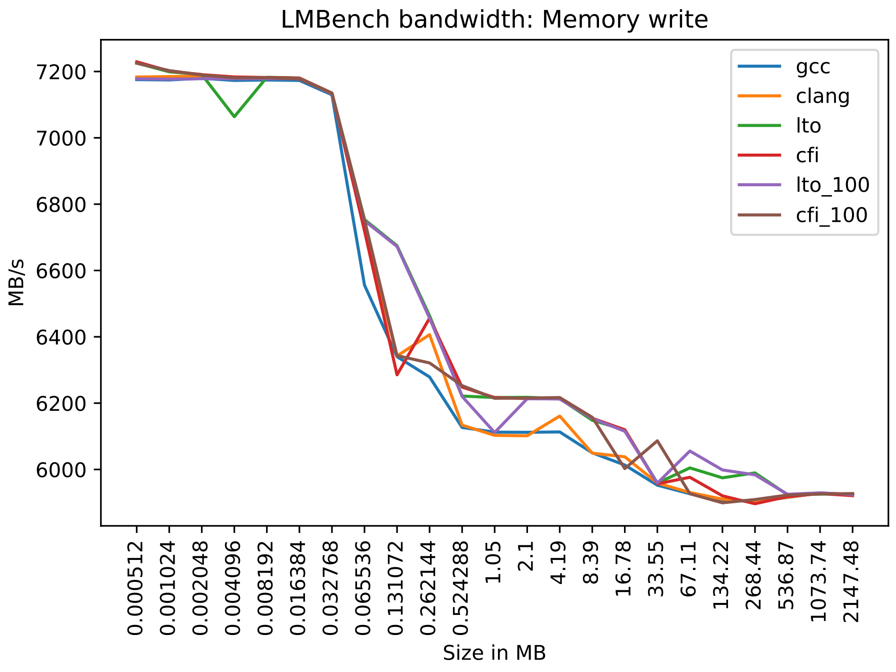


Figure 8.11: LMBench bandwidth memory write results

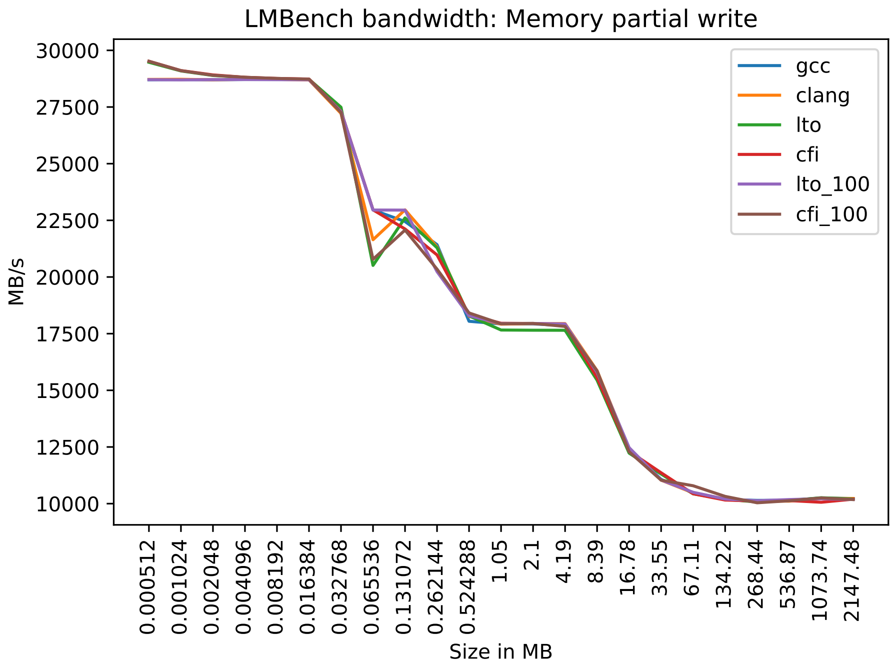


Figure 8.12: LMBench bandwidth memory partial write results

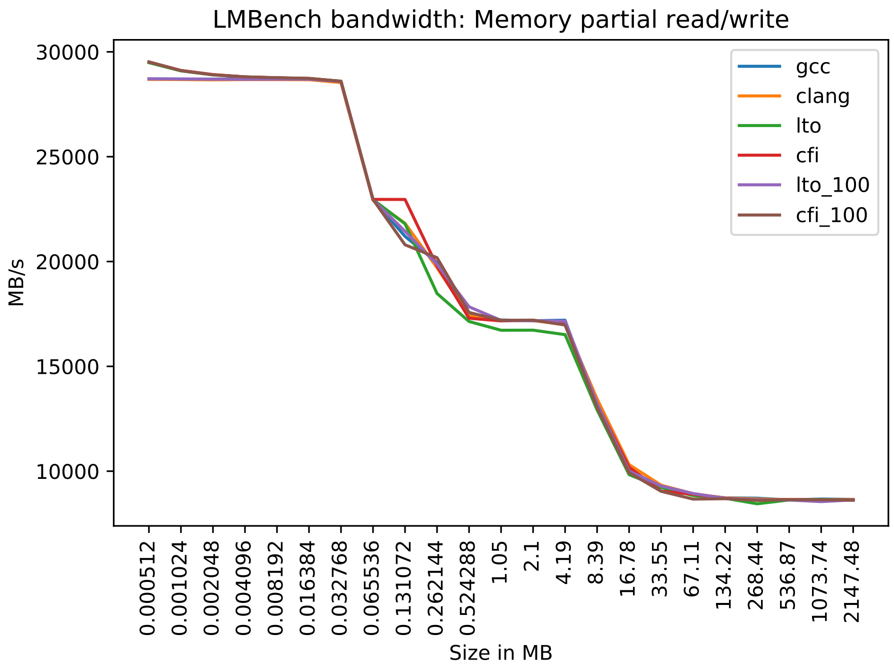


Figure 8.13: LMBench bandwidth memory partial read/write results

```
SYSCALL_DEFINE2(newstat, const char __user *, filename,
                struct stat __user *, statbuf)
{
        struct kstat stat;
        int error = vfs_stat(filename, &stat);

        if (error)
                return error;
        return cp_new_stat(&stat, statbuf);
}
```

Figure 8.14: `newstat()` definition in the Linux kernel

The entry point in the kernel images is `__x64_sys_newstat()`. For 32-bit syscalls, the entry point is `__ia32_sys_newstat()`. This function is just a thin wrapper around the real implementation, which is named `__do_sys_newstat()` in the gcc kernel, and `__se_sys_newstat()` in the kernels compiled with clang. The implementation examples from the different kernels have been converted from assembly to C for a more compact representation of the code. In addition, we have attempted to use code from the kernel sources to make it look more natural. The code for the gcc and clang kernels can be found in figure 8.15 and figure 8.16 respectively.

```
int __do_sys_newstat(const char *filename, struct stat *statbuf)
{
        int error;
        struct kstat stat;

        error = vfs_statx(AT_FDCWD, filename, AT_NO_AUTOMOUNT,
                          &stat, STATX_BASIC_STATS);
        if (error)
                return error;
        return cp_new_stat(&stat, statbuf);
}
```

Figure 8.15: `newstat()` from gcc kernel

```
int __se_sys_newstat(const char *filename, struct stat *statbuf)
{
        int error;
        struct kstat stat;
        struct stat tmp;

        error = vfs_statx(AT_FDCWD, filename, AT_NO_AUTOMOUNT,
                          &stat, STATX_BASIC_STATS);
        if (error)
                return error;

        if (!valid_dev(stat->dev) || !valid_dev(stat->rdev))
                return -EOVERFLOW;

        INIT_STRUCT_STAT_PADDING(tmp);
        tmp.st_dev = encode_dev(stat->dev);
        tmp.st_ino = stat->ino;
        if (sizeof(tmp.st_ino) < sizeof(stat->ino) && tmp.st_ino != stat->ino)
                return -EOVERFLOW;
        tmp.st_mode = stat->mode;
        tmp.st_nlink = stat->nlink;
        if (tmp.st_nlink != stat->nlink)
                return -EOVERFLOW;
        SET_UID(tmp.st_uid, from_kuid_munged(current_user_ns(), stat->uid));
        SET_GID(tmp.st_gid, from_kgid_munged(current_user_ns(), stat->gid));
        tmp.st_rdev = encode_dev(stat->rdev);
        tmp.st_size = stat->size;
        tmp.st_atime = stat->atime.tv_sec;
        tmp.st_mtime = stat->mtime.tv_sec;
        tmp.st_ctime = stat->ctime.tv_sec;
        tmp.st_blocks = stat->blocks;
        tmp.st_blksize = stat->blksize;
        return copy_to_user(statbuf, &tmp, sizeof(tmp)) ? -EFAULT : 0;
}
```

Figure 8.16: `newstat()` from clang kernels

The first thing to note is that the function is significantly smaller in the gcc kernel. The reason for this is that the call to `cp_new_stat()` has been inlined in the kernel compiled with clang. Other than this, the clang, LTO, and CFI kernels look the same. However, if we look into `vfs_statx()` we start to see more differences.

We will not display full code listings here, but rather we will discuss important differences between the different versions of the compiled function. The original code definition of `vfs_statx()` in the kernel looks like this:

```
1  int vfs_statx(int dfd, const char __user *filename, int flags,
2                struct kstat *stat, u32 request_mask)
3  {
4          struct path path;
5          int error = -EINVAL;
6          unsigned lookup_flags;
7
8          if (vfs_stat_set_lookup_flags(&lookup_flags, flags))
9                  return -EINVAL;
10 retry:
11         error = user_path_at(dfd, filename, lookup_flags, &path);
12         if (error)
13                 goto out;
14
15         error = vfs_getattr(&path, stat, request_mask, flags);
16         path_put(&path);
17         if (retry_estale(error, lookup_flags)) {
18                 lookup_flags |= LOOKUP_REVAL;
19                 goto retry;
20         }
21 out:
22         return error;
23 }
```

In the gcc kernel, no real differences between the binary and source code were found. `vfs_stat_set_lookup_flags()` on line 8 and `user_path_at()` on line 11 were both inlined, but these are marked as `static inline` in the kernel sources, so this is not a huge surprise. Clang inlined the call to `vfs_getattr()` as well. This is a very small function that is not used many places in the kernel, however, so it makes sense to inline it. The LTO kernel takes it one step further and inlines `user_path_at_empty()` as well. In the CFI kernel, however, the call to `user_path_at_empty()` is not inlined as it goes through a jump table.

`user_path_at_empty()` is defined in `fs/namei.c` while `vfs_statx()` is defined in `fs/stat.c`, so it makes sense that the clang kernel is not able to inline this call as it exists in another object file. The function is very small, so it makes sense that the LTO kernel inlines it across object files.

Recall that the LTOv2 kernel can inline larger functions across object file boundaries, so we expect to see more aggressive inlining in the LTOv2 kernel compared to the others. This is exactly what we observe, as the LTOv2 kernel inlines the call to `path_put()` as well. The CFIv2 kernel also inlines the call to `path_put()`, but the call to `user_path_at_empty()` still goes through a jump table and thus is not inlined.

Both the LTOv2 and CFIv2 kernels further inlines `vfs_getattr()`. This is how the function looks in C code:

```
int vfs_getattr(const struct path *path, struct kstat *stat,
                u32 request_mask, unsigned int query_flags)
{
        int retval;

        retval = security_inode_getattr(path);
        if (retval)
                return retval;
        return vfs_getattr_nosec(path, stat, request_mask, query_flags);
}
```

The function `security_inode_getattr()` is actually small enough to be inlined here. This function will use a function pointer to perform an indirect function call, so the CFIv2 function is slightly larger since it has to validate the function call. The following table compares the sizes of `vfs_statx()` in the different kernels.

kernel	size	percent increase
gcc	205	N/A
clang	264	28.78 %
CFI	264	28.78 %
LTO	293	42.93 %
LTOv2	381	85.85 %
CFIv2	398	94.15 %

Table 8.4: Size of `vfs_statx()`

An increase in code size does not mean that the number of instructions executed when running the function increases, however. When inlining functions, the function call overhead is removed, as there are no `call` instructions used, no need to pass arguments in registers, and no need for function prologues and epilogues. In addition, the compiler might be able to perform optimizations that were not possible before inlining the call, like smarter register usage and so on.

Consider the assembly code listing from the gcc kernel in figure 8.17. The code is from the `user_path_at_empty()` function, which was inlined on several kernel configurations. When this function is inlined, there is no need for the prologue, which contains roughly 10 instructions. The epilogue is also omitted, saving around 5 instructions. Although modern CPUs can execute billions of instructions per second, the number of instructions can have an impact if the number is big enough. So when the whole kernel is affected by aggressive inlining, the overall number of instructions executed to perform an action may decrease, leading to faster execution times. For these simple results, however, it seems like the performance may be negatively impacted by the increased code size, as the gcc kernel performed well for many of the benchmarks. The difference was miniscule, however, so maybe LMBench does not produce results that are accurate enough to really see any difference.

```
        push    rbp
        mov     rbp, rsp
        push    r14
        mov     r14, rcx
        push    r13
        mov     r13d, edi
        mov     rdi, rsi
        push    r12
        mov     r12d, edx
        mov     rdx, r8
        mov     esi, r12d
        call    getname_flags
        mov     rcx, r14
        mov     edx, r12d
        mov     edi, r13d
        mov     rsi, rax
        xor     r8d, r8d
        call    filename_lookup_0
        pop     r12
        pop     r13
        pop     r14
        pop     rbp
        retn
```

Figure 8.17: `user_path_at_empty()` assembly code from gcc kernel

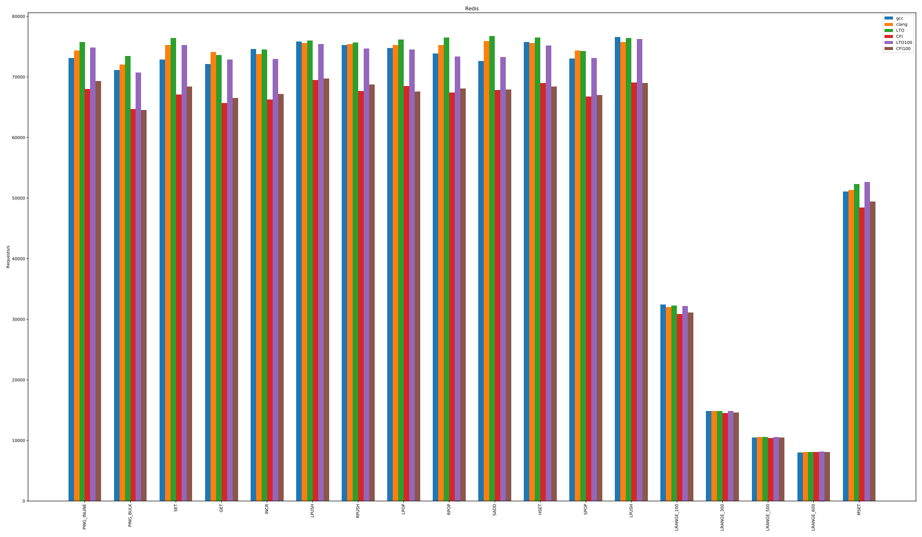


Figure 8.18: redis-benchmark results

8.6 Macro benchmarks

The following sections will describe the results from the macro benchmarks performed on x86_64.

8.6.1 Redis

In figure 8.18 we can see the results from running redis' benchmarking suite on the different kernels.

The Y axis on the graph measures requests per second, which means that higher numbers mean better performance. For this benchmark we can clearly see that CFI is the slowest for a majority of the benchmarks. gcc kernels perform a little better than clang on average, but the LTO kernel performs very well, beating the others in almost every single benchmark. One interesting thing to note is that LTO usually beats LTOv2 in these benchmarks, which may indicate that increased inlining may be slower for certain workloads.

kernel	PING_INLINE	PING_BULK	SET	GET	INCR	LPUSH	RPUSH	LPOP	RPOP	SADD	HSET	SPOP	LPUSH	LRANGE_100	LRANGE_300	LRANGE_500	LRANGE_600	MSET
gcc	3.56	3.23	4.89	2.74	0.0	0.15	0.61	1.90	3.52	5.76	0.99	1.78	0.0	0.0	0.07	0.86	2.71	2.94
clang	1.82	1.91	1.60	0.0	1.12	0.46	0.30	1.29	1.68	1.15	1.15	0.0	0.99	1.30	0.25	0.16	1.60	2.58
lto	0.0	0.0	0.0	0.67	0.15	0.0	0.0	0.0	0.0	0.0	0.0	0.07	0.15	0.58	0.0	0.19	1.08	0.53
cfi	11.43	13.51	13.98	12.81	12.53	9.34	11.80	11.20	13.46	13.20	10.86	11.29	10.86	5.22	2.36	1.31	1.43	8.57
lto100	1.21	3.82	1.53	1.70	2.24	0.76	1.29	2.28	4.20	4.83	1.76	1.63	0.38	0.84	0.25	0.0	0.0	0.0
cfi100	9.24	13.80	11.76	11.41	11.04	8.88	10.06	12.72	12.31	13.12	11.85	11.00	11.02	4.24	1.99	1.22	1.79	6.41

Table 8.5: Redis benchmark results

To see the difference between the benchmarks more clearly, we have plotted the difference in percent in table 8.5. The green cells represent the fastest benchmarks, the yellow ones are 0.1 to 4.9 % slower, and the red cells are more than 5 % slower. We can clearly see that the LTO kernel is the fastest overall. For the benchmarks where the LTO kernel is not the fastest, it is within 1 % of the fastest benchmark in all benchmarks except one.

8.6.2 ApacheBench

This section shows the results from running ApacheBench against nginx and Apache on the different kernels. Figure 8.19 shows the number of requests per second for nginx. Figure 8.20 shows the average request time for nginx. Figure 8.21 shows the number of requests per second for Apache, and figure 8.22 shows the average request time for Apache. The benchmarks are run with 1, 10, 20, and 30 concurrent connections, and each of these benchmarks are run 10 times. An average is calculated over these results.

In addition to the figures we have some tables showing the performance difference between the benchmarks. The best benchmarks are represented by green cells. The yellow cells represent benchmarks with a decreased performance of 0.1 to 4.9 %. The red cells represent benchmarks that have a slowdown of 5.0 % or more.

kernel	1	10	20	30
gcc	4.07	6.35	3.72	5.58
clang	1.52	8.19	3.37	3.09
lto	1.30	4.23	0.0	0.98
cfi	15.62	16.28	14.17	14.68
lto100	0.0	0.0	1.24	0.0
cfi100	14.70	17.11	17.12	17.36

Table 8.6: ApacheBench nginx requests/s

kernel	1	10	20	30
gcc	4.28	6.36	3.91	5.64
clang	1.52	8.90	3.68	3.06
lto	1.41	4.51	0.0	0.71
cfi	15.49	16.42	14.61	14.34
lto100	0.0	0.0	1.84	0.0
cfi100	14.54	17.46	17.49	17.27

Table 8.7: ApacheBench nginx time per request

kernel	1	10	20	30
gcc	6.50	4.10	5.06	3.67
clang	0.0	2.70	2.69	1.18
lto	2.19	0.0	3.73	1.16
cfi	17.75	13.08	13.93	12.23
lto100	13.80	0.26	0.0	0.0
cfi100	21.60	12.01	13.03	12.33

Table 8.8: ApacheBench apache2 requests/s

The redis benchmarks showed that the LTO kernel performed the best, while these benchmarks show that LTOv2 has the best overall performance. The CFI kernels see a significant performance drop here as well.

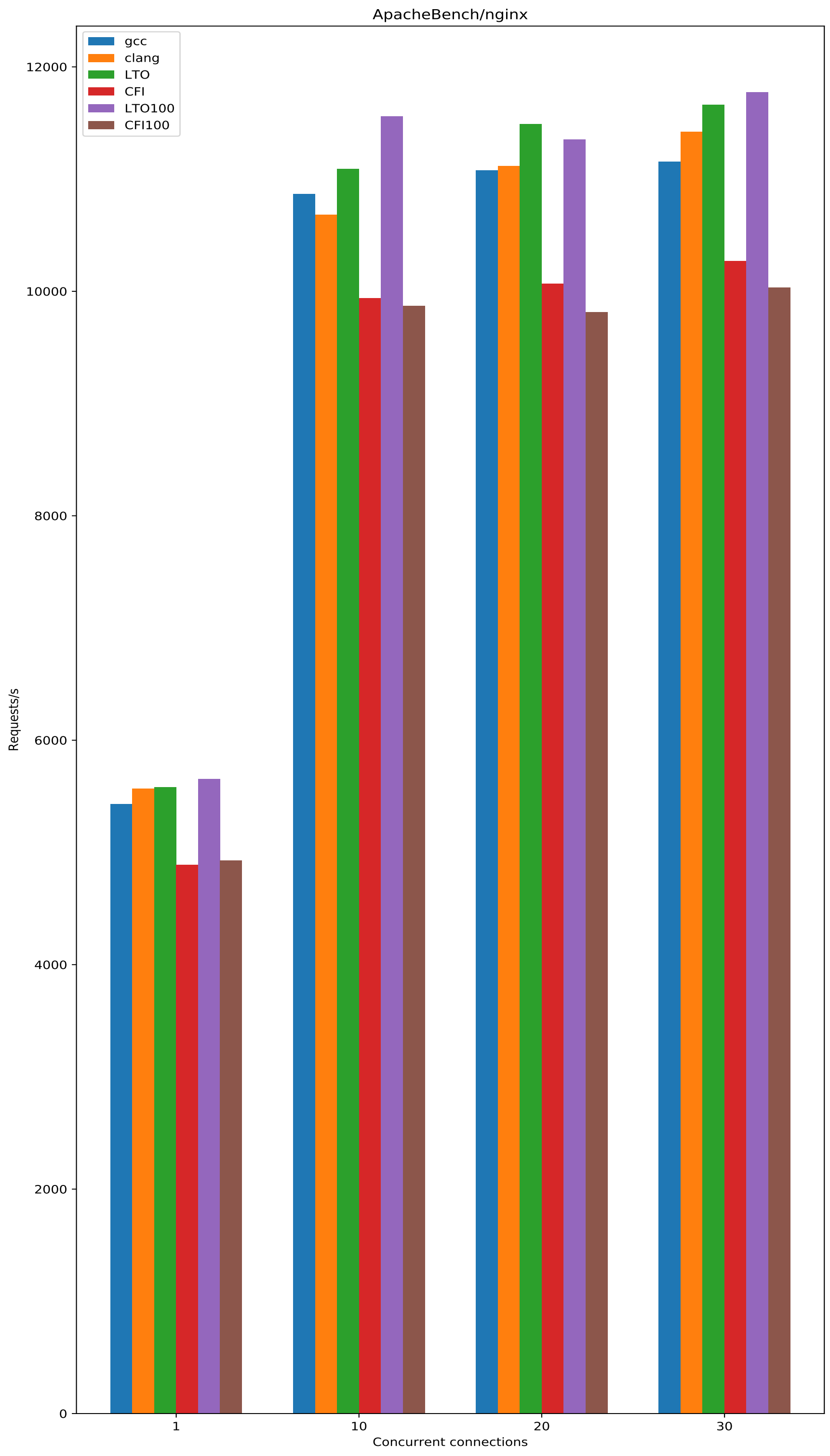


Figure 8.19: ApacheBench nginx requests/s

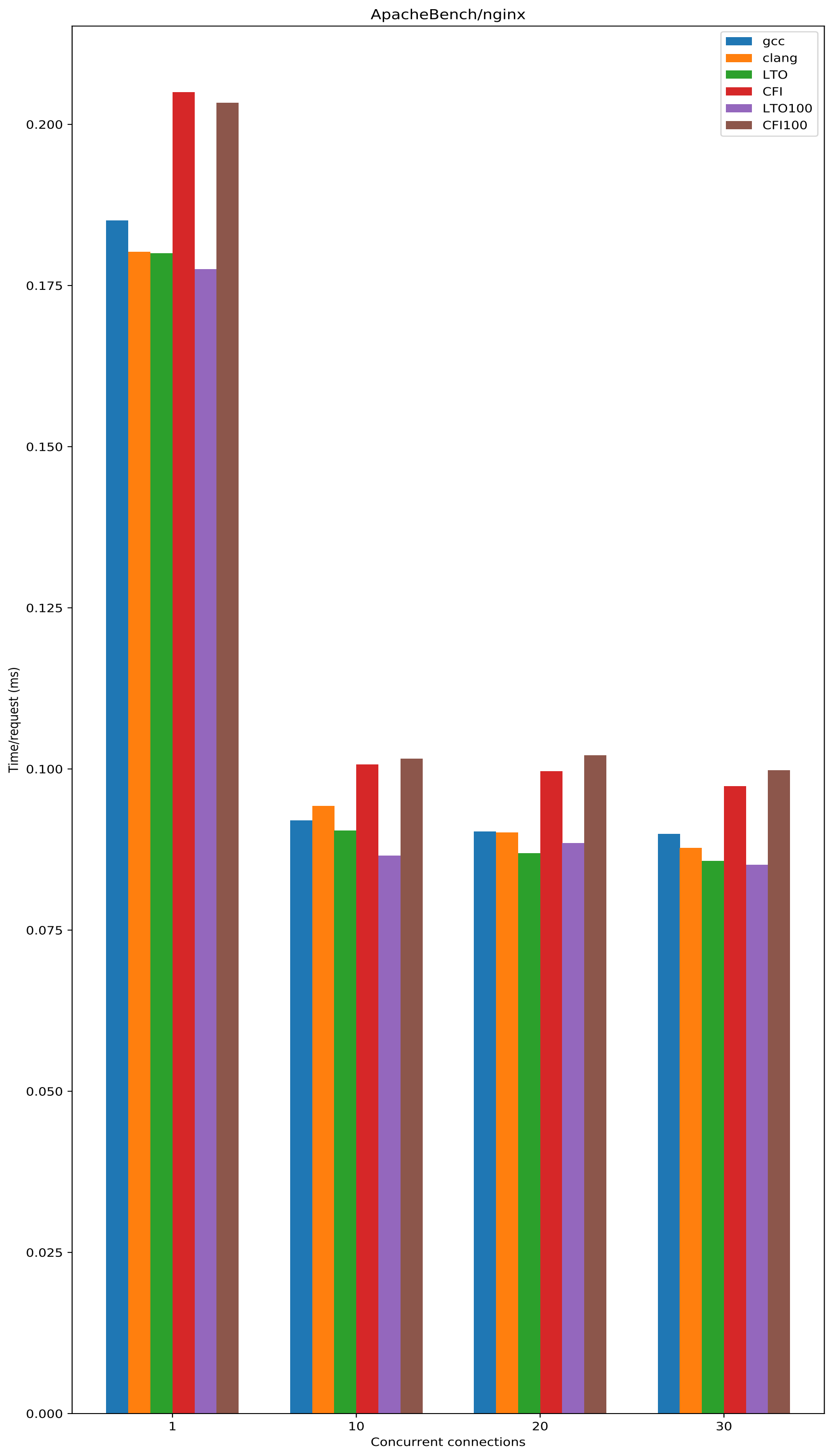


Figure 8.20: ApacheBench nginx time per request

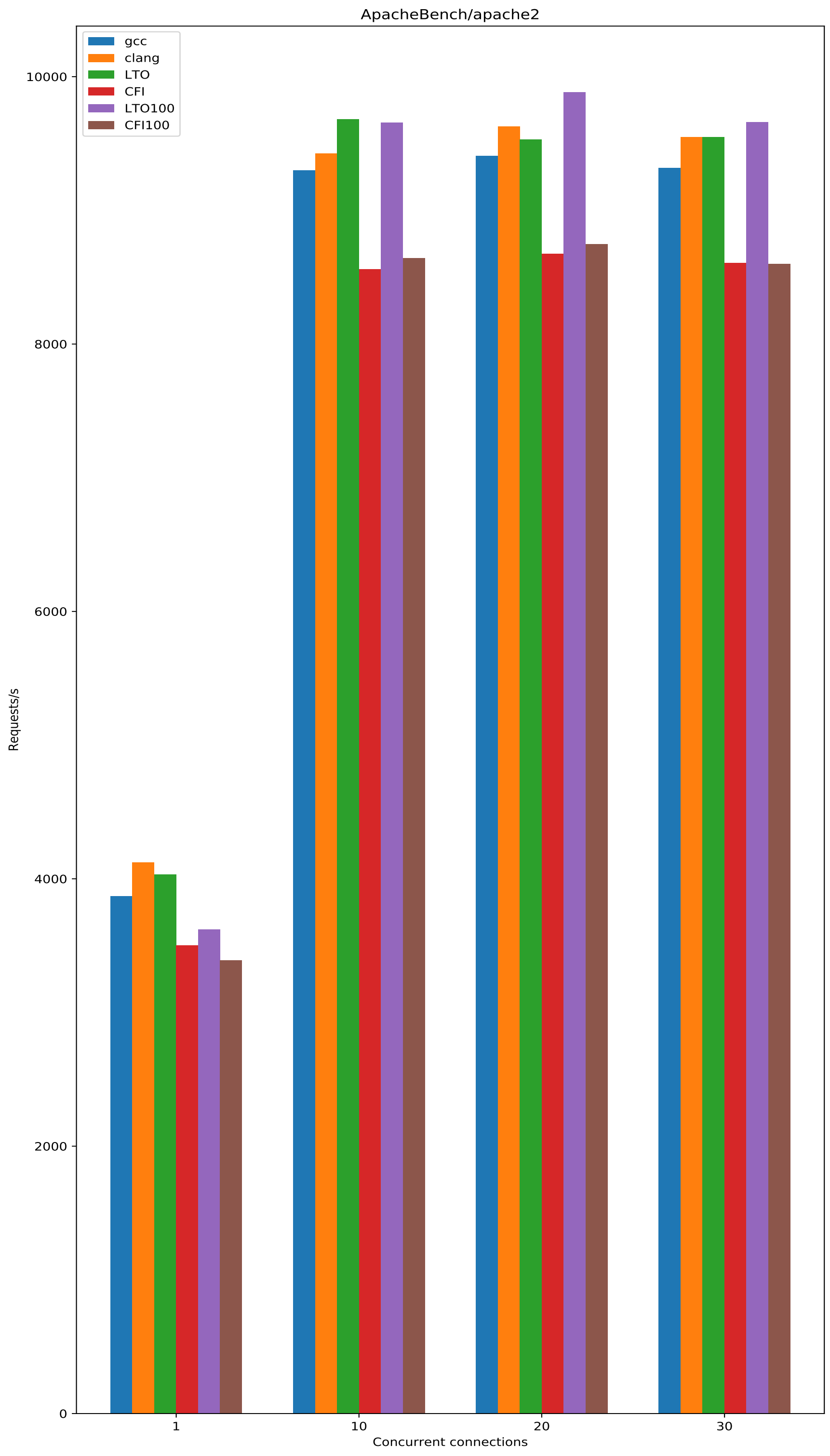


Figure 8.21: ApacheBench apache2 requests/s

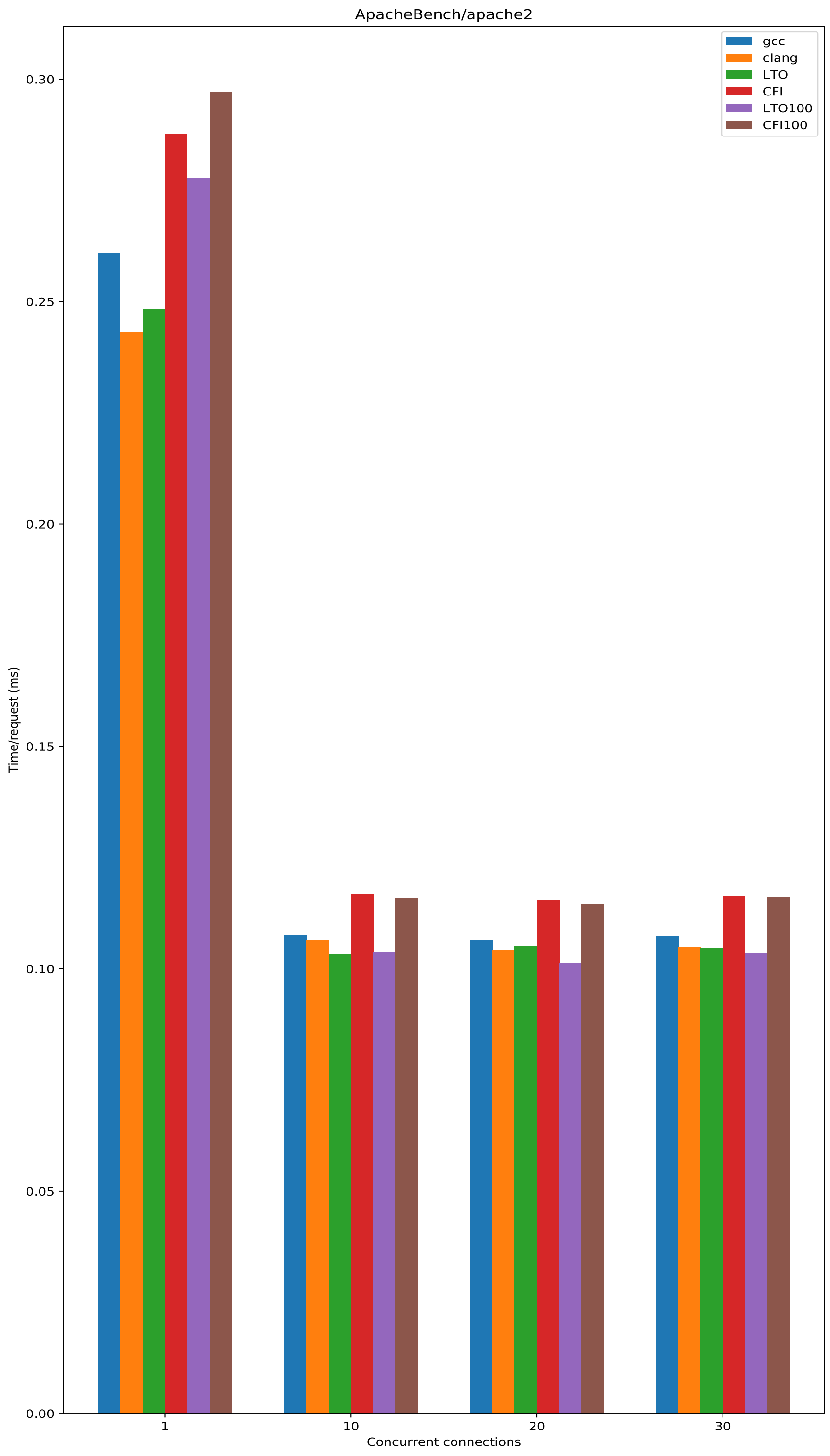


Figure 8.22: ApacheBench apache2 time per request

kernel	1	10	20	30
gcc	7.24	4.16	5.03	3.57
clang	0.0	3.00	2.76	1.16
lto	2.10	0.0	3.75	1.06
cfi	18.26	13.07	13.82	12.26
lto100	14.19	0.39	0.0	0.0
cfi100	22.16	12.20	12.93	12.16

Table 8.9: ApacheBench apache2 time per request

8.6.3 WireGuard

In this section we will present the results of running the WireGuard benchmark detailed in section 7.6. In figure 8.23 we have the UDP throughput results for WireGuard. In figures 8.24 and 8.25 we have the results for TCP send and receive, respectively.

kernel	udp_ipv4_jumbo_ipv4	udp_ipv4_jumbo_ipv6	udp_ipv4_normal_ipv4	udp_ipv4_normal_ipv6	udp_ipv6_jumbo_ipv4	udp_ipv6_jumbo_ipv6	udp_ipv6_normal_ipv4	udp_ipv6_normal_ipv6
gcc	17.73	2.90	11.90	0.04	0.0	0.0	0.0	2.32
clang	9.46	0.0	5.13	2.78	13.85	11.38	2.80	4.19
lto	0.76	6.75	3.59	2.17	4.15	6.79	0.89	0.0
cfi	17.96	10.89	8.36	2.89	15.69	3.91	9.84	6.03
lto100	0.0	0.89	0.0	0.0	0.44	2.44	2.70	4.85
cfi100	14.29	20.66	9.67	5.82	10.42	25.68	8.55	10.16

Table 8.10: WireGuard UDP send bandwidth

kernel	tcp_ipv4_jumbo_ipv4	tcp_ipv4_jumbo_ipv6	tcp_ipv4_normal_ipv4	tcp_ipv4_normal_ipv6	tcp_ipv6_jumbo_ipv4	tcp_ipv6_jumbo_ipv6	tcp_ipv6_normal_ipv4	tcp_ipv6_normal_ipv6
gcc	0.0	3.62	0.0	0.0	0.67	3.34	0.65	0.79
clang	0.95	3.45	3.01	0.54	3.75	3.40	2.04	1.82
lto	1.69	2.39	3.09	1.16	0.0	0.0	0.0	0.0
cfi	3.22	2.04	3.46	3.72	2.62	7.26	5.65	4.73
lto100	0.15	0.0	0.31	0.57	2.21	5.62	0.13	1.73
cfi100	13.66	16.15	7.68	4.71	13.12	15.56	7.65	6.31

Table 8.11: WireGuard TCP send bandwidth

kernel	tcp_ipv4_jumbo_ipv4	tcp_ipv4_jumbo_ipv6	tcp_ipv4_normal_ipv4	tcp_ipv4_normal_ipv6	tcp_ipv6_jumbo_ipv4	tcp_ipv6_jumbo_ipv6	tcp_ipv6_normal_ipv4	tcp_ipv6_normal_ipv6
gcc	0.0	3.62	0.0	0.0	0.67	3.35	0.66	0.79
clang	0.95	3.45	3.02	0.54	3.75	3.40	2.04	1.83
lto	1.70	2.39	3.09	1.17	0.0	0.0	0.0	0.0
cfi	3.23	2.04	3.47	3.73	2.62	7.26	5.65	4.74
lto100	0.15	0.0	0.32	0.56	2.21	5.62	0.13	1.73
cfi100	13.66	16.15	7.68	4.71	13.12	15.57	7.65	6.32

Table 8.12: WireGuard TCP receive bandwidth

Some of the benchmarks are named *jumbo*, while others are named *normal*. This refers to the *maximum transmission unit* (MTU) used when sending the packets. MTU relates to the maximum size of packets sent on the network. Thus, increasing the MTU means that more data can be sent with each network packet. This explains why the jumbo benchmarks have a higher throughput than the normal ones.

The same results are also plotted as tables like we have done with the other macro benchmarks. The UDP send bandwidth benchmarks can be found in table 8.10. The TCP send and receive bandwidth benchmarks can be found in tables 8.11 and 8.12 respectively.

The results for the WireGuard benchmarks are less clear than the other macro benchmarks we have performed. We could probably have run these

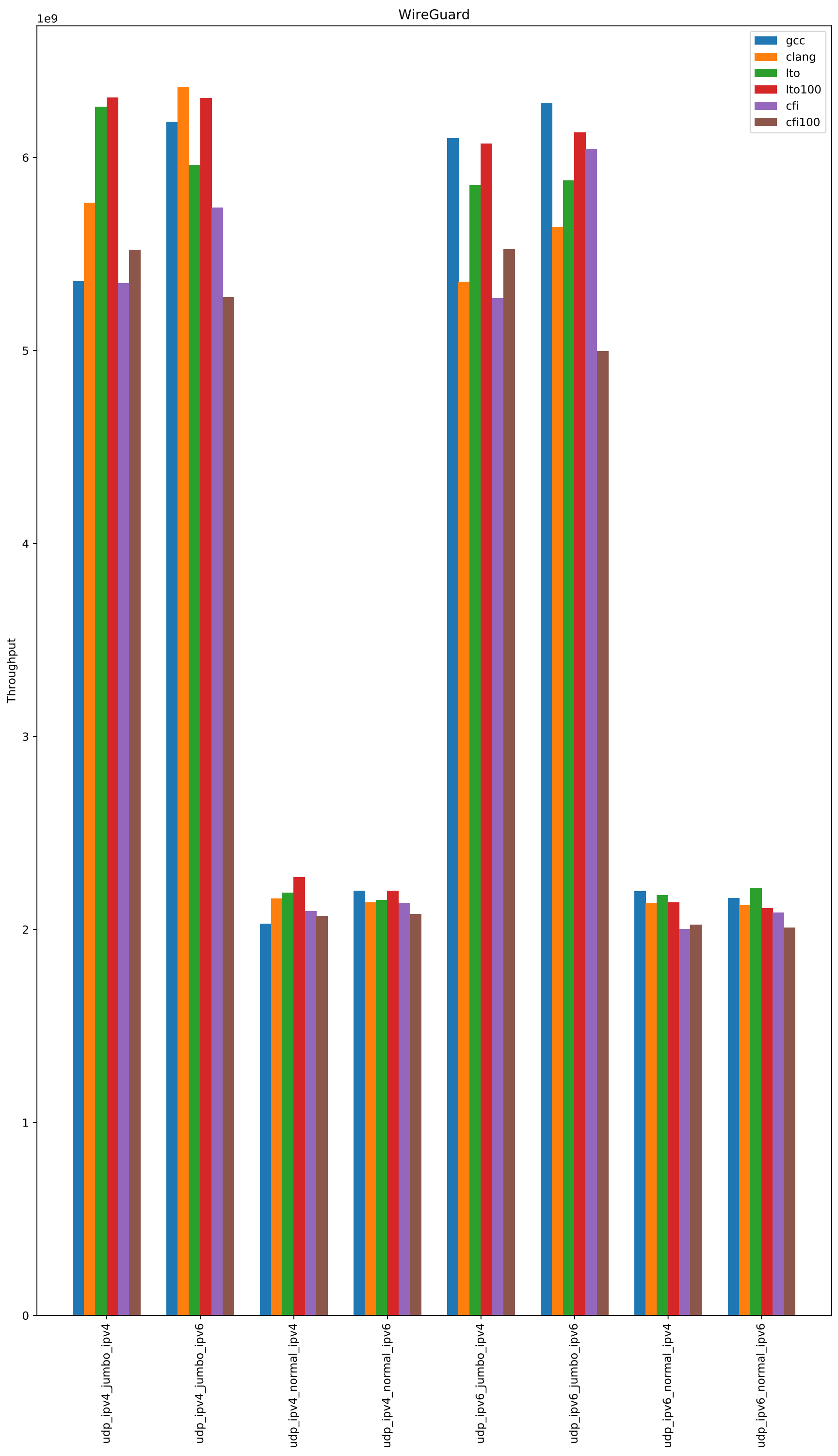


Figure 8.23: WireGuard UDP send bandwidth

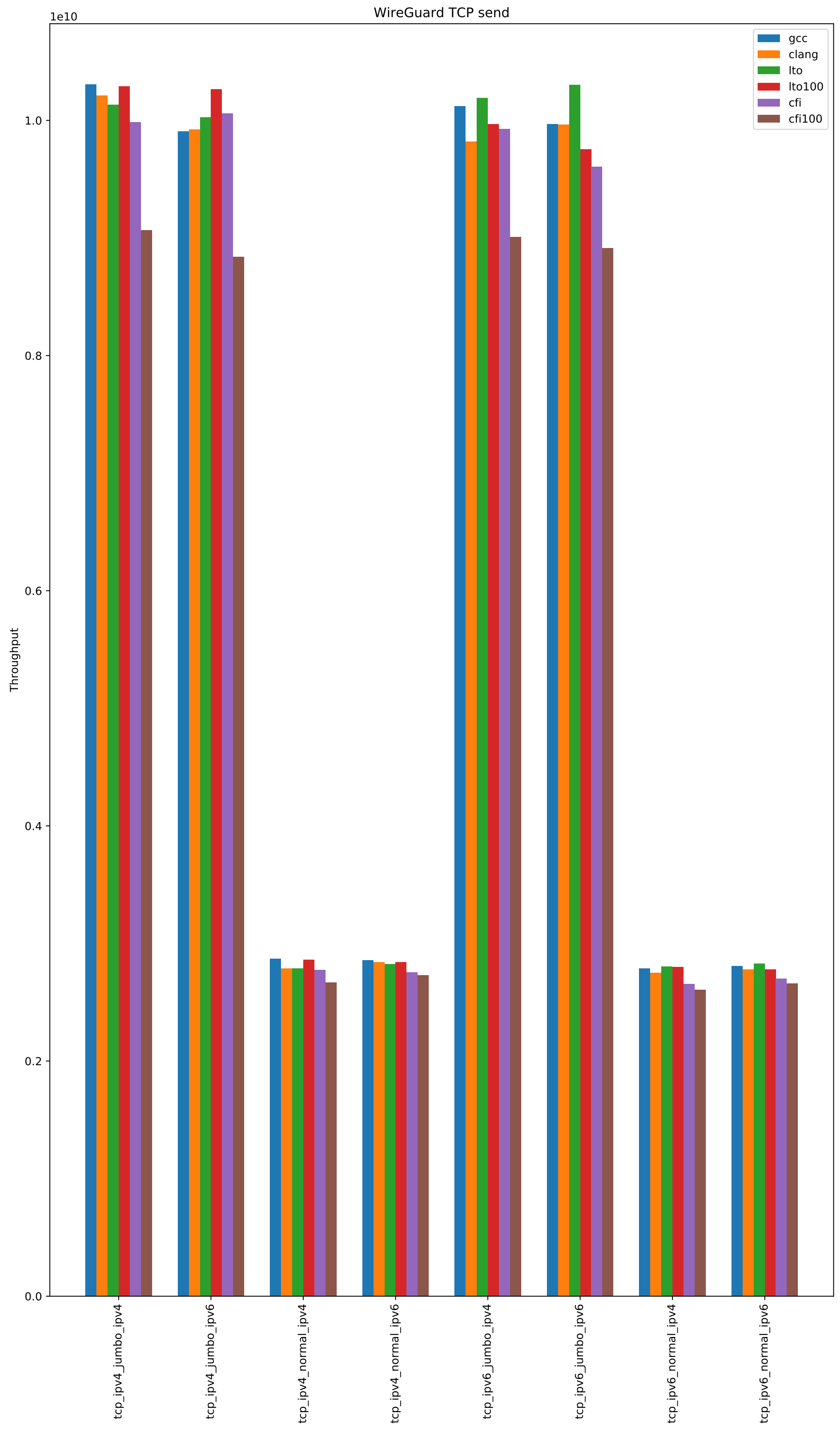


Figure 8.24: WireGuard TCP send bandwidth

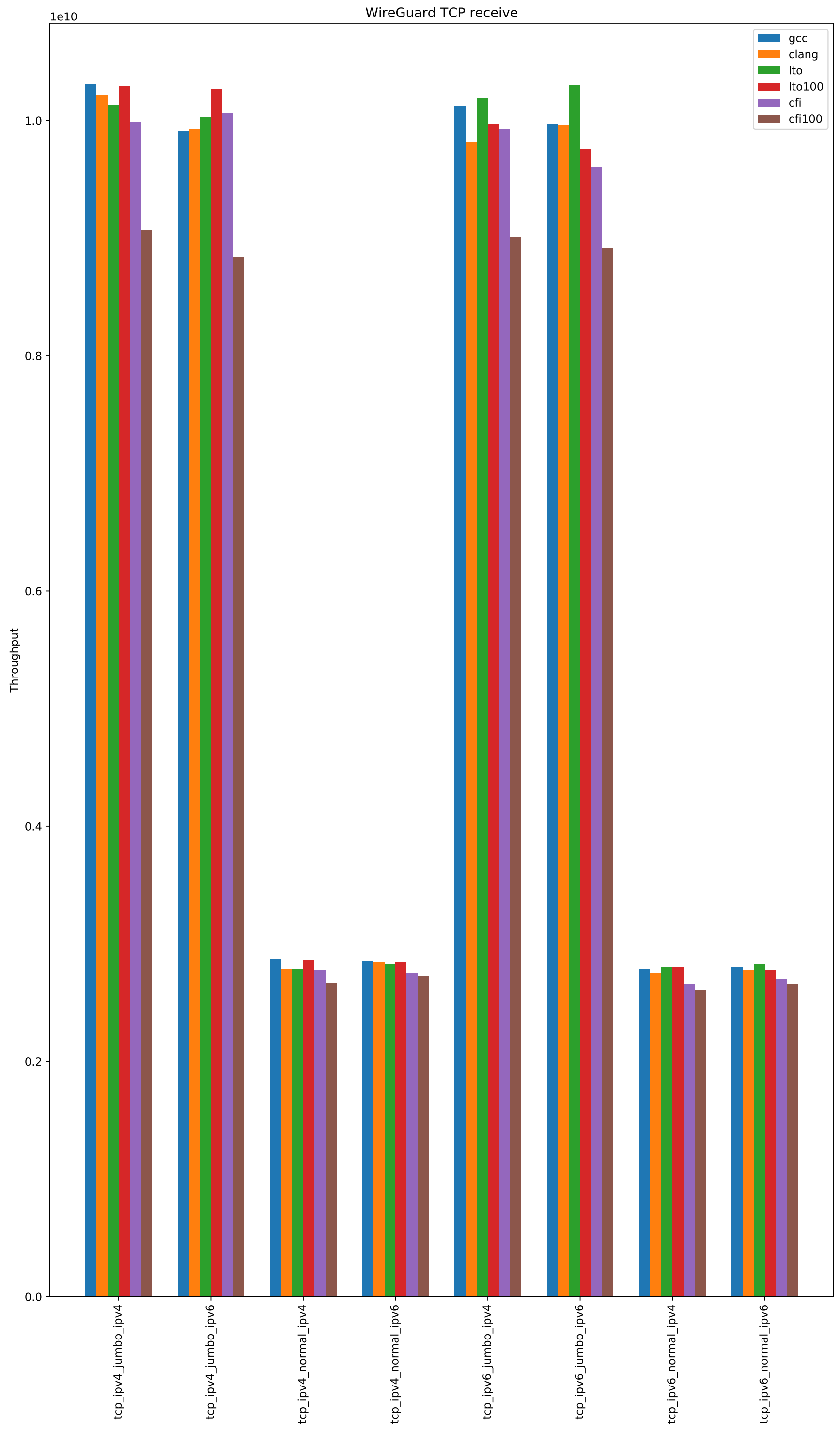


Figure 8.25: WireGuard TCP receive bandwidth

benchmarks several times and computed the average to get more accurate results. We still observe that the CFI kernels perform worse than the other kernels by between 2.04 % and 25.68 %.

8.7 Android benchmarks

Unfortunately the results from the benchmarking tools used on the HiKey 960 were not reliable enough to provide any good insight into the performance impact of LTO, CFI, and SCS. Every run using the same benchmarks on the same kernel would provide very different results, even though the number of iterations for the benchmarks were set to a high number. At most, 128 iterations were used for the binder/hwbinder throughput benchmarks.

Running the benchmarks for a single kernel with 128 iterations for each benchmark took around 2 hours, making it very cumbersome to try different approaches for improving the results. It is not clear what causes the results to vary so much, but one strategy could probably be to run the benchmarks several times, rebooting the HiKey between every run and then calculating the average over all the results. However, this would have taken a very long time if all the kernels were to be benchmarked. In addition, Google only saw a 1-2 % performance hit, so it might not be possible to notice any difference on certain hardware.

8.8 Summary

It is hard to say if clang's CFI implementation is worth using on x86_64 because of the performance impact. Although some exploitation scenarios are made more difficult for attackers, only the forward-edge of the control flow graph is protected. Coupled with SCS it would probably be more attractive. When CET-enabled processors from Intel arrive on the market it would be interesting to compare the overhead of enabling that compared to clang's CFI implementation. The security benefit is not the same, however, since CFI checks the function signature, while CET still allows function pointer hijacking as long as the destination is the start of a valid function. With CET we will also get backward-edge CFI, which, coupled with forward-edge CFI provides a pretty good defense.

Enabling LTO showed a significant increase in performance on x86_64, and it would be interesting to see if more Linux distributions start picking up clang and LTO. Although there were some issues with PTI and CFI, no issues were encountered while using a kernel with LTO enabled. It would be interesting to test the robustness of the kernel in a production environment to see if it still holds up, or if there are any unknown issues.

Using LTO and CFI really increases compilation time, especially on weaker hardware. Thus these features are not very well suited during rapid development. LTO and CFI should probably only be enabled once the kernel is ready for release, when extra compilation time is not detrimental for developer productivity.

Binary size might be an issue in certain environments, like embedded devices. Here, gcc might be the preferred option as it was able to produce a smaller kernel. However, on x86_64 the size increase is probably negligible. When using LTO/CFI with an inline limit of 100, the kernel increased 30.87 % in size. From 8 MB to 11 MB. 3 MB is definitely not a problem for a normal laptop or desktop computer as they usually have several hundred GB of storage. However, the increased kernel size will also affect memory usage of the system. For embedded devices with limited amounts of RAM this may become an issue.

Overall the results show that LTO can provide a performance boost for certain workloads. Some workloads saw an increased performance when using LTO with an increased inlining limit, while others saw a performance drop compared to less aggressive inlining. There are definitely potential for more research in this area to find a good balance for the inlining limit.

8.9 Future work

One goal should be to get LTO support upstreamed. When that is done, support for CFI can be added on top of it. Upstreaming the code means that more people have the chance to review and provide feedback on the changes. The patchsets available look more or less ready for upstreaming so this might happen very soon. Support for LTO, CFI, and SCS is already merged into the Android kernel sources, so all vendors relying on Android can enable it for their devices. Many other features from Android has ended up in the upstream kernel before so it is not unlikely that it eventually will land there. However, as of May 2020, it does not look like any vendors except Google have enabled any of these features.

It would be beneficial to test more versions of clang, but a lot of bug fixes are included in clang 11, which makes it possible to compile the upstream Linux kernel without any workarounds. Using older versions of clang/LLVM does not make that much sense at this time, since we have to go out of our way to make everything work correctly. It would be more beneficial to wait for newer versions to be released, and see if the LTO/CFI/SCS implementations are improved. Although other projects have successfully used clang versions earlier than 11 to build the kernel, LTO and CFI complicates the process.

The current state of CFI on x86_64 is probably not ready for production systems yet. Mainly because the kernel will panic on CFI violations, and there are still a lot of false positives due to function signature mismatches, like the example in 6.3.3. Before CFI can be applied to production systems, work has to be done to make sure that there are very few false positives left. An alternative could be to enable permissive mode for a while to weed out false positives. However, with permissive mode none of the security properties of CFI are enforced, and an attacker can hijack function pointers without having to bypass the function signature checking. While writing this thesis, a kernel with CFI enabled was used for most of the time to see that it is suitable for normal workloads. No instability or crashes were observed while using the CFI kernel.

The CFI kernel would not boot with PTI enabled, so making sure that LTO/CFI is compatible with PTI would be beneficial, as PTI provides protection against certain side-channels vulnerabilities.

It would be interested to look further into the effects increased kernel and code size has on the instruction cache. Increasing the code size might have a negative impact on cache locality, leading to decreased performance in some circumstances. There is almost no difference in performance when increasing the LTO inlining limit, so it is probably not worth increasing the limit anyways. Knowing the impact on the CPU cache is still interesting, and this might vary a lot between different CPUs as well, making it an interesting research topic.

Appendix A

Scripts

A.1 Linux kernel installation script

```bash
#!/bin/bash

set -eux

KERNEL_VERSION=5.7.0-rc2

if [ "$1" == "gcc" ]; then
        echo "[+] installing gcc kernel"
        OUTPUT_DIR=gcc_out
        sudo make INSTALL_MOD_STRIP=1 O=$OUTPUT_DIR modules_install
        sudo make O=$OUTPUT_DIR install
        sudo depmod
elif [ "$1" == "cfi" ]; then
        echo "[+] installing LTO/CFI kernel"
        OUTPUT_DIR=cfi_out
elif [ "$1" == "lto" ]; then
        echo "[+] installing LTO kernel"
        OUTPUT_DIR=lto_out
elif [ "$1" == "clang" ]; then
        echo "[+] installing clang kernel"
        OUTPUT_DIR=clang_out
else
        echo "[-] specify a kernel type!"
        exit
fi

# strip from binutils doesn't really like modules compiled with clang
# and especially not modules compiled with LTO, so we have to strip the
# modules manually before generating initrd
sudo make O=$OUTPUT_DIR modules_install
sudo find /lib/modules/$KERNEL_VERSION-$1+ -name \*.ko -exec \
         llvm-strip-9 --strip-debug {} +
sudo make O=$OUTPUT_DIR install
sudo depmod
```

A.2 Linux kernel build script

```
#!/bin/bash

set -eux

if [ "$1" == "gcc" ]; then
        echo "[+] vanilla kernel with gcc"
        OUTPUT_DIR=gcc_out
        mkdir -p $OUTPUT_DIR

        make defconfig HOSTCC=gcc CC=gcc O=gcc_out

        cat ubuntu_config > $OUTPUT_DIR/.config
        scripts/config --file $OUTPUT_DIR/.config \
                --set-str CONFIG_LOCALVERSION "-gcc" \

        # to work around __memcat_p bug
        #scripts/config --file $OUTPUT_DIR/.config -e CONFIG_STM

        make HOSTCC="gcc" CC="gcc" O=$OUTPUT_DIR olddefconfig
        time make -j8 HOSTCC="gcc" CC="gcc" O=$OUTPUT_DIR 2>&1 \
                | tee build_results_$1.txt
        exit 0
fi

if [ "$1" == "cfi" ]; then
        echo "[+] LTO/CFI kernel"
        OUTPUT_DIR=cfi_out
elif [ "$1" == "lto" ]; then
        echo "[+] LTO kernel"
        OUTPUT_DIR=lto_out
elif [ "$1" = "full_lto" ]; then
        echo "[+] full LTO kernel"
        OUTPUT_DIR=full_lto_out
elif [ "$1" == "clang" ]; then
        echo "[+] clang kernel"
        OUTPUT_DIR=clang_out
else
        echo "[-] specify a kernel type!"
        exit
fi

mkdir -p $OUTPUT_DIR

make defconfig HOSTCC=clang-11 HOSTLD=ld.lld-11 \
        CC=clang-11 HOSTCC=clang-11 LD=ld.lld-11 O=$OUTPUT_DIR

cat ubuntu_config > $OUTPUT_DIR/.config
# to work around __memcat_p bug
#scripts/config --file $OUTPUT_DIR/.config -e CONFIG_STM
make HOSTCC="clang-11" CC="clang-11" O=$OUTPUT_DIR olddefconfig

if [ "$1" == "cfi" ]; then
        echo "[+] LTO/CFI kernel"
        OUTPUT_DIR=cfi_out
        mkdir -p $OUTPUT_DIR
        # Enable LTO and CFI.
        scripts/config --file $OUTPUT_DIR/.config \
                -e CONFIG_LTO \
                -e CONFIG_THINLTO \
                -d CONFIG_LTO_NONE \
                -e CONFIG_LTO_CLANG \
                -e CONFIG_CFI_CLANG \
                -e CONFIG_CFI_PERMISSIVE \
                -e CONFIG_CFI_CLANG_SHADOW \
                --set-str CONFIG_LOCALVERSION "-cfi"
elif [ "$1" == "lto" ]; then
        echo "[+] LTO kernel"
        OUTPUT_DIR=lto_out
        mkdir -p $OUTPUT_DIR
        scripts/config --file $OUTPUT_DIR/.config \
                -e CONFIG_LTO \
                -e CONFIG_THINLTO \
                -d CONFIG_LTO_NONE \
                -e CONFIG_LTO_CLANG \
                -d CONFIG_CFI_CLANG \
                --set-str CONFIG_LOCALVERSION "-lto"
elif [ "$1" = "full_lto" ]; then
        echo "[+] full LTO kernel"
        scripts/config --file $OUTPUT_DIR/.config \
                -e CONFIG_LTO \
                -d CONFIG_LTO_NONE \
                -d CONFIG_THINLTO \
                -e CONFIG_LTO_CLANG \
                -d CONFIG_CFI_CLANG \
                --set-str CONFIG_LOCALVERSION "-full_lto"
elif [ "$1" == "clang" ]; then
        echo "[+] clang kernel"
        OUTPUT_DIR=clang_out
        mkdir -p $OUTPUT_DIR
        scripts/config --file $OUTPUT_DIR/.config \
                -e CONFIG_LTO_NONE \
                --set-str CONFIG_LOCALVERSION "-clang"
fi

```

```
time make -j8 HOSTCC=clang-11 HOSTLD=ld.lld-11 \
                                CC=clang-11 LD=ld.lld-11 O=$OUTPUT_DIR 2>&1 \
                                | tee build_results_$1.txt
```

A.3 Android Linux kernel build script

Kernel customizations were performed as described in the official Android documentation [4]. A local `build.config` was created with `POST_DEFCONFIG_CMDS` defined to run a script that enables and disables configuration options. The different versions of the customization function can be found below.

```
# clang
function update_debug_config_clang() {
    ${KERNEL_DIR}/scripts/config --file ${OUT_DIR}/.config \
         -d LTO \
         -d LTO_CLANG \
         -d CFI \
         -d CFI_PERMISSIVE \
         -d CFI_CLANG \
         -d SHADOW_CALL_STACK \
         --set-str CONFIG_LOCALVERSION "-clang"
    (cd ${OUT_DIR} && \
     make O=${OUT_DIR} $archsubarch CC=${CC} CROSS_COMPILE=${CROSS_COMPILE} olddefconfig)
}

# LTO
function update_debug_config_lto() {
    ${KERNEL_DIR}/scripts/config --file ${OUT_DIR}/.config \
         -e LTO \
         -e LTO_CLANG \
         -d CFI \
         -d CFI_CLANG \
         -d SHADOW_CALL_STACK \
         --set-str CONFIG_LOCALVERSION "-lto"
    (cd ${OUT_DIR} && \
     make O=${OUT_DIR} $archsubarch CC=${CC} CROSS_COMPILE=${CROSS_COMPILE} olddefconfig)
}

# CFI
function update_debug_config_cfi() {
    ${KERNEL_DIR}/scripts/config --file ${OUT_DIR}/.config \
         -e LTO \
         -e LTO_CLANG \
         -e CFI \
         -e CFI_PERMISSIVE \
         -e CFI_CLANG \
         -d SHADOW_CALL_STACK \
         --set-str CONFIG_LOCALVERSION "-cfi"
    (cd ${OUT_DIR} && \
     make O=${OUT_DIR} $archsubarch CC=${CC} CROSS_COMPILE=${CROSS_COMPILE} olddefconfig)
}

# ShadowCallStack
function update_debug_config_scs() {
    ${KERNEL_DIR}/scripts/config --file ${OUT_DIR}/.config \
         -e LTO \
         -e LTO_CLANG \
         -e CFI \
         -e CFI_PERMISSIVE \
         -e CFI_CLANG \
        -e SHADOW_CALL_STACK \
        --set-str CONFIG_LOCALVERSION "-scs"
    (cd ${OUT_DIR} && \
     make O=${OUT_DIR} $archsubarch CC=${CC} CROSS_COMPILE=${CROSS_COMPILE} olddefconfig)
}
```

A.4 Docker

The following Dockerfile was used to build all the Linux kernel images.

```
FROM clangbuiltlinux/debian

RUN apt update && apt install -y gcc vim
COPY ccache.conf /root/.ccache/ccache.conf
RUN ln -s /usr/bin/llvm-ar-11 /usr/bin/llvm-ar && \
        ln -s /usr/bin/llvm-nm-11 /usr/bin/llvm-nm && \
        ln -s /usr/bin/clang-11 /usr/bin/clang && \
        ln -s /usr/bin/llvm-dis-11 /usr/bin/llvm-dis
# for make menuconfig support
RUN apt install -y libncurses-dev
# needed by the packaging targets for the Linux kernel, like tarbz2-pkg
RUN apt install -y cpio
# install packages required by the bindeb-pkg target
#RUN apt install -y build-essential devscripts fakeroot kmod rsync

ENTRYPOINT [ "/bin/bash" ]
```

With the following ccache config file for better incremental compilation performance.

```
cache_dir = /opt/ccache
max_size = 10.0G
```

`ccache` was not used when measuring compilation time.

A.5 Benchmarking script

The following scripts were used to run all the micro/macro benchmarks on x86_64.

```
#!/bin/bash

# stop some services that might be running
sudo systemctl stop gdm
sudo systemctl stop ssh
sudo systemctl stop redis
sudo systemctl stop nginx
sudo systemctl stop apache2

sudo lmbench-run

sudo systemctl start redis-server
redis-benchmark > redis_results_$(uname -r).txt

# run ApacheBench with Apache and nginx
python3 ab_to_pdf.py bench
```

And the following script was used to run all benchmarks on arm64.

```
#!/system/bin/sh

NAME=$1

# make all CPUs run at maximum freqency
for i in `seq 0 7`; do
    echo performance > /sys/devices/system/cpu/cpu$i/cpufreq/scaling_governor
done

# hwbinder throughput
./libhwbinder_benchmark --benchmark_out_format=json \
                        --benchmark_repetitions=64 \
                        --benchmark_report_aggregates_only=true \
                        --benchmark_out=hwbinder_throughput_$NAME.txt

# binder throughput
./libbinder_benchmark --benchmark_out_format=json \
                      --benchmark_repetitions=64 \
                      --benchmark_report_aggregates_only=true \
                      --benchmark_out=binder_throughput_$NAME.txt

# hwbinder latency
./libhwbinder_latency -i 10000 -pair 3 > hwbinder_latency_$NAME.txt

# binder latency
./schd-dbg -i 10000 -pair 3 > binder_latency_$NAME.txt
```

A.6 Python benchmarking scripts

The following script was used to convert the results from `redis-benchmark` to graphs:

```
#!/usr/bin/env python3
import os
import sys
import re
import matplotlib.pyplot as plt
import numpy as np


def parse(filename):
    with open(filename, "r") as f:
```

```python
            data = f.readlines()
    names = []
    values = []

    for i in range(len(data)):
        line = data[i]
        if line == "\n":
            continue

        if not line.startswith("====== "):
            continue

        # we found a new benchmark!
        name = line.split(" ")[1]
        names.append(name)

    for i in range(len(data)):
        line = data[i]
        if "requests per second" not in line:
            continue
        values.append(float(line.split(" ")[0]))

    return (names, values)


def main():
    results_path = "."
    files = os.listdir(results_path)
    results = {}

    for f in files:
        if not "redis_results_5.7.0-rc2" in f or f.startswith("."):
            continue
        print("parsing file {}...".format(f))
        pattern = r"redis_results_5.7.0-rc2-([a-z0-9]+)\+.txt"
        name = re.match(pattern, f).group(1)
        results[name] = parse("./" + f)

    gcc = results["gcc"][1]
    clang = results["clang"][1]
    lto = results["lto"][1]
    cfi = results["cfi"][1]
    lto100 = results["lto100"][1]
    cfi100 = results["cfi100"][1]

    labels = results["gcc"][0]

    print(labels)
    print(gcc)
    out = """\\begin{table*}[!htbp]
    \\begin{adjustbox}{width=1\\textwidth}
    \\centering
    \\begin{tabular}{@{}llllllllllllllllll@{}}
        kernel"""
    for label in labels:
        out += " & {}".format(label.replace("_", "\\_"))
    out += " \\\\\n"
    out += "\\midrule\n"

    targets = [ "gcc", "clang", "lto", "cfi", "lto100", "cfi100" ]
    fixed_results = {}

    for target in targets:
        fixed_results[target] = []

    # calculate best result
    # y axis is requests/s, so higher numbers are better
    for i in range(len(labels)):
        res = []
        for target in targets:
            res.append(results[target][1][i])

        # calculate the best result
        best = max(res)

        for i in range(len(res)):
            val = res[i]
            if val == best:
                res[i] = 0.0
            else:
                res[i] = (float(best - val) / val) * 100
        print(res)

        for i in range(len(targets)): #target in targets:
            target = targets[i]
            fixed_results[target].append(res[i])

    for target in targets:
        out += "\t\t{}".format(target)

        for result in fixed_results[target]:
            if result == 0.0:
                out += " & \\cellcolor{{green!50}}0.0".format(result)
            elif result > 5.0:
                out += " & \\cellcolor{{red!50}}{:.2f}".format(result)
            else:
                out += " & \\cellcolor{{yellow!50}}{:.2f}".format(result)
        out += " \\\\\n"
    out += "\\bottomrule\n"
    out += "\\end{tabular}\n"
    out += "\\end{adjustbox}\n"
    out += "\\caption{Redis benchmark results}\n"
    out += "\\label{fig:redis_results_table}\n"
    out += "\\end{table*}\n"
    print(out)

    x = np.arange(len(labels))
    width = 0.12

    fig, ax = plt.subplots()

    rects1 = ax.bar(x - (width * 2.5), gcc, width, label="gcc")
    rects2 = ax.bar(x - (width * 1.5), clang, width, label="clang")
    rects3 = ax.bar(x - (width * 0.5), lto, width, label="LTO")
    rects4 = ax.bar(x + (width * 0.5), cfi, width, label="CFI")
    rects5 = ax.bar(x + (width * 1.5), lto100, width, label="LTO100")
    rects6 = ax.bar(x + (width * 2.5), cfi100, width, label="CFI100")

    ax.set_ylabel("Requests/s")
    ax.set_title("Redis")
    ax.set_xticks(x)
    ax.set_xticklabels(labels)
    ax.legend()
    plt.xticks(rotation=90)

    fig.savefig("redis_results.pdf", bbox_inches="tight")


if __name__ == "__main__":
    main()
```

And the following script was used to both run benchmarks and graph results from ApacheBench:

```
#!/usr/bin/env python3
import os
import re
import matplotlib.pyplot as plt
import numpy as np
import sys


def parse(filename):
    with open(filename, "r") as f:
        data = f.read()
    res = {}

    rps = float(re.search(r"Requests per second:[\s]+([0-9]+\.[0-9]+)",
                          data).group(1))

    pattern = r"Time per request:[\s]+([0-9]+\.[0-9]+) \[ms\] \(mean, across"
    pattern += r" all concurrent requests\)"
    tpr = float(re.search(pattern, data).group(1))
    concurrency = int(re.search(r"Concurrency Level:[\s]+([0-9]+)",
                                data).group(1))

    print("Requests per second: {}".format(rps))
    print("Time per request:    {}".format(tpr))
    print("Concurrency Level:   {}".format(concurrency))

    return (concurrency, rps, tpr)


def run_benchmark(name, concurrency, n):
    cmd = "ab -n 1000 -c {0} http://127.0.0.1/ > ab/ab_results_{2}_$(uname "
    cmd += "-r)_{0}_{1}.txt"
    cmd = cmd.format(concurrency, n, name)
    os.system(cmd)


def plot_reqs(name, values):
    labels = [ "1", "10", "20", "30" ]
    x = np.arange(len(labels))
    width = 0.12
    fig, ax = plt.subplots()

    if len(values["gcc"]["request"]) != 0:
        rects1 = ax.bar(x - (width * 2.5), values["gcc"]["request"],
                        width, label="gcc")
    if len(values["clang"]["request"]) != 0:
        rects2 = ax.bar(x - (width * 1.5), values["clang"]["request"],
                        width, label="clang")
    if len(values["lto"]["request"]) != 0:
        rects3 = ax.bar(x - (width * 0.5), values["lto"]["request"],
                        width, label="LTO")
    if len(values["cfi"]["request"]) != 0:
        rects4 = ax.bar(x + (width * 0.5), values["cfi"]["request"],
                        width, label="CFI")
    if len(values["lto100"]["request"]) != 0:
        rects5 = ax.bar(x + (width * 1.5), values["lto100"]["request"],
                        width, label="LTO100")
    if len(values["cfi100"]["request"]) != 0:
        rects6 = ax.bar(x + (width * 2.5), values["cfi100"]["request"],
                        width, label="CFI100")

    ax.set_ylabel("Requests/s")
    ax.set_xlabel("Concurrent connections")
    ax.set_title("ApacheBench/{}".format(name))
    ax.set_xticks(x)
    ax.set_xticklabels(labels)
    ax.legend()
```

```python

    fig.tight_layout()
    fig.savefig("ab_results_requests_{}.pdf".format(name),
                bbox_inches="tight")
    generate_table(name, values, labels, "request")


def plot_time(name, values):
    labels = [ "1", "10", "20", "30" ] #, "Time per request" ]
    x = np.arange(len(labels))
    width = 0.12
    fig, ax = plt.subplots()

    if len(values["gcc"]["request"]) != 0:
        rects1 = ax.bar(x - (width * 2.5), values["gcc"]["time"],
                        width, label="gcc")
    if len(values["clang"]["request"]) != 0:
        rects2 = ax.bar(x - (width * 1.5), values["clang"]["time"],
                        width, label="clang")
    if len(values["lto"]["request"]) != 0:
        rects3 = ax.bar(x - (width * 0.5), values["lto"]["time"],
                        width, label="LTO")
    if len(values["cfi"]["request"]) != 0:
        rects4 = ax.bar(x + (width * 0.5), values["cfi"]["time"],
                        width, label="CFI")
    if len(values["lto100"]["request"]) != 0:
        rects5 = ax.bar(x + (width * 1.5), values["lto100"]["time"],
                        width, label="LTO100")
    if len(values["cfi100"]["request"]) != 0:
        rects6 = ax.bar(x + (width * 2.5), values["cfi100"]["time"],
                        width, label="CFI100")

    ax.set_ylabel("Time/request (ms)")
    ax.set_xlabel("Concurrent connections")
    ax.set_title("ApacheBench/{}".format(name))
    ax.set_xticks(x)
    ax.set_xticklabels(labels)
    ax.legend()

    fig.tight_layout()
    fig.savefig("ab_results_time_{}.pdf".format(name), bbox_inches="tight")
    generate_table(name, values, labels, "time")


def generate_table(name, values, labels, bench_type):
    targets = [ "gcc", "clang", "lto", "cfi", "lto100", "cfi100" ]
    out = """\\begin{table*}[!htbp]
    \\centering
    \\begin{tabular}{@{}lllllllllllllllll@{}}
        kernel"""
    for label in labels:
        out += " & {}".format(label.replace("_", "\\_"))
    out += " \\\\\n"
    out += "\\midrule\n"

    fixed_results = {}

    for target in targets:
        fixed_results[target] = []

    for i in range(len(labels)):
        res = []
        for target in targets:
            res.append(values[target][bench_type][i])

        # calculate the best result
        if bench_type == "time":
            best = min(res)
        else:
            best = max(res)

        for i in range(len(res)):
            val = res[i]
            if val == best:
                res[i] = 0.0
            else:
                if bench_type == "time":
                    res[i] = (float(val - best) / best) * 100
                else:
                    res[i] = (float(best - val) / val) * 100

        for i in range(len(targets)): #target in targets:
            target = targets[i]
            fixed_results[target].append(res[i])

    for target in targets:
        out += "\t\t{}".format(target)

        for result in fixed_results[target]:
            if result == 0.0:
                out += " & \\cellcolor{{green!50}}0.0".format(result)
            elif result > 5.0:
                out += " & \\cellcolor{{red!50}}{:.2f}".format(result)
            else:
                out += " & \\cellcolor{{yellow!50}}{:.2f}".format(result)
        out += " \\\\\n"
    out += "\\bottomrule\n"
    out += "\\end{tabular}\n"
    if bench_type == "time":
        description = "time per request"
    else:
        description = "requests/s"
    out += "\\caption{{ApacheBench {} {}}}\n".format(name, description)
    if bench_type == "time":
        suffix = "tps"
    else:
        suffix = "rps"
    out += "\\label{{table:ab_{}_{}}}\n".format(name, suffix)
    out += "\\end{table*}\n"

    print(out)


def main():
    results_path = "./ab"
    files = os.listdir(results_path)
    results = {}

    if len(sys.argv) != 2:
        print("Usage: {} [bench|pdf]".format(sys.argv[0]))
        sys.exit()

    if sys.argv[1] == "bench":
        os.system("sudo systemctl stop nginx")
        os.system("sudo systemctl stop apache2")

        os.system("sudo systemctl start nginx")
        # run the benchmark n times
        for concurrency in [1, 10, 20, 30]:
            for i in range(10):
                run_benchmark("nginx", concurrency, i)
        os.system("sudo systemctl stop nginx")

        os.system("sudo systemctl start apache2")
        # run the benchmark n times
        for concurrency in [1, 10, 20, 30]:
            for i in range(10):
                run_benchmark("apache2", concurrency, i)
        os.system("sudo systemctl stop apache2")

        sys.exit()

    if sys.argv[1] != "pdf":
        print("invalid command!")
        sys.exit()

    for webserver in [ "apache2", "nginx" ]:
        results[webserver] = {}
        for name in ["gcc","clang","lto","cfi","lto100","cfi100"]:
            results[webserver][name] = {}
```

```
                for concurrency in [1,10,20,30]:
                    results[webserver][name][concurrency] = list()

    for f in files:
        if f.startswith("."):
            continue
        print("parsing file {}...".format(f))
        pattern = r"ab_results_([a-z0-9]+)_([0-9]\.[0-9]+\.[0-9]-rc[0-9]-"
        pattern += r"([a-z0-9]+))\+_[0-9]+_[0-9]+.txt"
        m = re.match(pattern, f)
        webserver = m.group(1)
        version = m.group(2)
        name = m.group(3)

        concurrency, rps, tpr = parse("{}/{}".format(results_path, f))
        results[webserver][name][concurrency].append((rps, tpr))

    values = {}
    # now calculate the mean of all the results for each concurrency level
    for webserver in [ "apache2", "nginx" ]:
        for name in ["gcc","clang","lto","cfi","lto100","cfi100"]:
            values[name] = {}
            values[name]["request"] = list()
            values[name]["time"] = list()
            if name not in results[webserver]:
                continue
            for concurrency in [1,10,20,30]:
                res = results[webserver][name][concurrency]
                if len(res) == 0:
                    break
                avg_rps = sum(map(lambda x : x[0], res))/len(res)
                avg_tpr = sum(map(lambda x : x[1], res))/len(res)
                values[name]["request"].append(avg_rps)
                values[name]["time"].append(avg_tpr)
        # plot requests per second and time per request
        plot_reqs(webserver, values)
        plot_time(webserver, values)


if __name__ == "__main__":
    main()
```

To plot the results from lmbench, the following script was used:

```
#!/usr/bin/env python3
import os
import sys
import re
import matplotlib.pyplot as plt
import numpy as np

```

```

def convert(filename):
    with open(filename, encoding="utf8", errors="ignore") as f:
        data = f.read()

    m = re.search(r"\[RELEASE: (.*)\]", data)
    release = m.group(1)

    m = re.search("Simple syscall: ([0-9]+\.[0-9]+) microseconds", data)
    simple_syscall = m.group(1)
    m = re.search("Simple read: ([0-9]+\.[0-9]+) microseconds", data)
    simple_read = m.group(1)
    m = re.search("Simple write: ([0-9]+\.[0-9]+) microseconds", data)
    simple_write = m.group(1)
    m = re.search("Simple stat: ([0-9]+\.[0-9]+) microseconds", data)
    simple_stat = m.group(1)
    m = re.search("Simple fstat: ([0-9]+\.[0-9]+) microseconds", data)
    simple_fstat = m.group(1)
    m = re.search("Simple open/close: ([0-9]+\.[0-9]+) microseconds", data)
    simple_open_close = m.group(1)

    m = re.search("Select on 10 fd's: ([0-9]+\.[0-9]+) microseconds", data)
    select_10 = m.group(1)
    m = re.search("Select on 100 fd's: ([0-9]+\.[0-9]+) microseconds", data)
    select_100 = m.group(1)
    m = re.search("Select on 250 fd's: ([0-9]+\.[0-9]+) microseconds", data)
    select_250 = m.group(1)
    m = re.search("Select on 500 fd's: ([0-9]+\.[0-9]+) microseconds", data)
    select_500 = m.group(1)

    m = re.search("Select on 10 tcp fd's: ([0-9]+\.[0-9]+) microseconds",
                  data)
    select_tcp_10 = m.group(1)
    m = re.search("Select on 100 tcp fd's: ([0-9]+\.[0-9]+) microseconds",
                  data)
    select_tcp_100 = m.group(1)
    m = re.search("Select on 250 tcp fd's: ([0-9]+\.[0-9]+) microseconds",
                  data)
    select_tcp_250 = m.group(1)
    m = re.search("Select on 500 tcp fd's: ([0-9]+\.[0-9]+) microseconds",
                  data)
    select_tcp_500 = m.group(1)


    m = re.search("Signal handler installation: ([0-9]+\.[0-9]+) microseconds",
                  data)
    signal_handler_install = m.group(1)
    m = re.search("Signal handler overhead: ([0-9]+\.[0-9]+) microseconds",
                  data)
    signal_handler_overhead = m.group(1)
    m = re.search("Protection fault: ([0-9]+\.[0-9]+) microseconds", data)
    protection_fault = m.group(1)
    m = re.search("Pipe latency: ([0-9]+\.[0-9]+) microseconds", data)
    pipe_latency = m.group(1)
    m = re.search("AF_UNIX sock stream latency: ([0-9]+\.[0-9]+) microseconds",
                  data)
    af_unix_stream_lat = m.group(1)
    m = re.search("Process fork\+exit: ([0-9]+\.[0-9]+) microseconds",
                  data)
    proc_fork_exit = m.group(1)
    m = re.search("Process fork\+execve: ([0-9]+\.[0-9]+) microseconds", data)
    proc_fork_execve = m.group(1)
    m = re.search("Process fork\+/bin/sh -c: ([0-9]+\.[0-9]+) microseconds",
                  data)
    proc_fork_bin_sh = m.group(1)

    bw_names = [ "read", "read open2close", "Mmap read",
                 "Mmap read open2close", "Memory bzero", "Memory read",
                 "Memory partial read", "Memory write", "Memory partial write",
                 "Memory partial read/write" ]
    # get bandwidth results
    bw_res = {}
    data = data.split("\n")
    for i in range(len(data)):
        line = data[i]
        if line == "\n":
            continue

        if not "bandwidth" in line:
            continue

        # we found a new benchmark!
        name = line.replace("\"", "").replace(" bandwidth", "")
        if name not in bw_names:
            continue

        # there are 23 different values
        bw_res[name] = {}
        for j in range(i + 1, i + 1 + 23):
            tmp = data[j].split(" ")
            mb = float(tmp[0])
            mbs = float(tmp[1])
            bw_res[name][mb] = mbs

    name = release.split("-")[-1][:-1]
    res = { name: { "simple": {
            "Simple syscall": float(simple_syscall),
            "Simple read": float(simple_read),
            "Simple write": float(simple_write),
            "Simple stat": float(simple_stat),
            "Simple fstat": float(simple_fstat),
            "Simple open/close": float(simple_open_close),
            "Signal handler installation": float(signal_handler_install),
            "Signal handler overhead": float(signal_handler_overhead),
            "Protection fault": float(protection_fault),
            "Pipe latency": float(pipe_latency),
            "AF\\_UNIX sock stream latency": float(af_unix_stream_lat),
        }, "select": {
            "Select on 10 fd's": float(select_10),
            "Select on 100 fd's": float(select_100),
            "Select on 250 fd's": float(select_250),
            "Select on 500 fd's": float(select_500),
            "Select on 10 tcp fd's": float(select_tcp_10),
            "Select on 100 tcp fd's": float(select_tcp_100),
            "Select on 250 tcp fd's": float(select_tcp_250),
            "Select on 500 tcp fd's": float(select_tcp_500),
        }, "process": {
            "Process fork+exit": float(proc_fork_exit),
            "Process fork+execve": float(proc_fork_execve),
            "Process fork+/bin/sh -c": float(proc_fork_bin_sh),
            }
        }
    }

    return res, { name: bw_res }


def print_plot(results, bandwidth):
    if "gcc" not in results:
        print("Missing results for gcc!")
    if "clang" not in results:
        print("Missing results for clang!")
    if "lto" not in results:
        print("Missing results for lto!")
    if "cfi" not in results:
        print("Missing results for cfi!")
    if "lto100" not in results:
        print("Missing results for lto100!")
    if "cfi100" not in results:
        print("Missing results for cfi100!")

    gcc = results["gcc"]
    clang = results["clang"]
    lto = results["lto"]
    cfi = results["cfi"]
    lto_100 = results["lto100"]
    cfi_100 = results["cfi100"]

    for bench_type in gcc.keys():
        labels = gcc[bench_type].keys()

        x = np.arange(len(labels))
        width = 0.12

        gcc_values = list(gcc[bench_type].values())
        clang_values = list(clang[bench_type].values())
        lto_values = list(lto[bench_type].values())
        cfi_values = list(cfi[bench_type].values())
        cfi100_values = list(cfi_100[bench_type].values())
        lto100_values = list(lto_100[bench_type].values())

        fig, ax = plt.subplots()
        rects1 = ax.bar(x - (width * 2.5), gcc_values, width, label="gcc")
        rects2 = ax.bar(x - (width * 1.5), clang_values, width, label="clang")
        rects3 = ax.bar(x - (width * 0.5), lto_values, width, label="lto")
        rects4 = ax.bar(x + (width * 0.5), cfi_values, width, label="cfi")
        rects5 = ax.bar(x + (width * 1.5), lto100_values, width,
                        label="lto_100")
        rects6 = ax.bar(x + (width * 2.5), cfi100_values, width,
                        label="cfi_100")

        ax.set_ylabel("microseconds")
        ax.set_title("LMBench")
        ax.set_xticks(x)
        ax.set_xticklabels(labels)
        ax.legend()
        plt.xticks(rotation=90)

        fig.tight_layout()

        fig.savefig("lmbench_results_{}.pdf".format(bench_type),
                    bbox_inches="tight")

    gcc = bandwidth["gcc"]
    clang = bandwidth["clang"]
    lto = bandwidth["lto"]
    cfi = bandwidth["cfi"]
    lto_100 = bandwidth["lto100"]
    cfi_100 = bandwidth["cfi100"]
    for bench_type in gcc.keys():
        labels = gcc[bench_type].keys()
        labels = list(gcc[bench_type].keys())
        x = np.arange(len(labels))

        gcc_values = list(gcc[bench_type].values())
        clang_values = list(clang[bench_type].values())
        lto_values = list(lto[bench_type].values())
        cfi_values = list(cfi[bench_type].values())
        lto_100_values = list(lto_100[bench_type].values())
        cfi_100_values = list(cfi_100[bench_type].values())

        fig, ax = plt.subplots()
        t = list(map(float, labels))
        ax.plot(x, gcc_values, label="gcc")
        ax.plot(x, clang_values, label="clang")
        ax.plot(x, lto_values, label="lto")
        ax.plot(x, cfi_values, label="cfi")
        ax.plot(x, lto_100_values, label="lto_100")
        ax.plot(x, cfi_100_values, label="cfi_100")

        ax.set_ylabel("MB/s")
        ax.set_xlabel("Size in MB")
        ax.set_title("LMBench bandwidth: {}".format(bench_type))
        ax.set_xticks(x)
        ax.set_xticklabels(labels)
        plt.xticks(rotation=90)
        ax.legend()

        fig.tight_layout()

        filename = bench_type.lower().replace(" ", "_").replace("/", "_")
        fig.savefig("lmbench_results_bandwidth_{}.pdf".format(filename),
                    bbox_inches="tight")


def main():
    results_path = "/var/lib/lmbench/results/x86_64-linux-gnu/"
    files = os.listdir(results_path)
    results = {}
    bandwidth = {}
    for f in files:
        res, bw_res = convert(results_path + "/" + f)
        results.update(res)
        bandwidth.update(bw_res)

    print_plot(results, bandwidth)


if __name__ == "__main__":
    main()
```

To plot the results from the Android benchmarks, the following Python script was used:

```
#!/usr/bin/env python3
import json
import sys
import matplotlib.pyplot as plt
import numpy as np


```

```python
targets = [ "clang", "lto", "lto100", "cfi", "cfi100", "scs", "scs100" ]


def plot_binder_throughput():
    results = {}
    for target in targets:
        results[target] = {}
        filename = "binder_throughput_{}.txt".format(target)
        with open(filename, "r") as f:
            data = json.load(f)
        benchmarks = data["benchmarks"]

        for bench in benchmarks:
            name = bench["name"]
            if "_mean" not in name:
                continue
            size = int(name.split("/")[1].split("_")[0])
            iterations = bench["iterations"]
            time = bench["real_time"]

            # b/ns
            # for an approximate result, multiply the data transfer
            # rate value by 1.074
            throughput = (size * iterations) / time
            print(name)
            print("({} * {}) / {}".format(size, iterations, time))
            print("throughput: {:.2f} b/ns".format(throughput))
            print("throughput: {:.2f} Gb/s".format(throughput * 1.074))
            if not name in results[target]:
                results[target][name] = []
            results[target][name].append(throughput)

    # calculate average
    for key, value in results.items():
        res = results[key]

        for bench_name, bench_val in value.items():
            avg = sum(bench_val) / len(bench_val)
            value[bench_name] = avg

    clang = results["clang"].values()
    lto = results["lto"].values()
    lto100 = results["lto100"].values()
    cfi = results["cfi"].values()
    cfi100 = results["cfi100"].values()
    scs = results["scs"].values()
    scs100 = results["scs100"].values()

    labels = [ x.split("/")[1] for x in results["clang"].keys() ]

    x = np.arange(len(labels))
    width = 0.12

    fig, ax = plt.subplots()

    rects1 = ax.bar(x - (width * 3.0), clang, width, label="clang")
    rects2 = ax.bar(x - (width * 2.0), lto, width, label="LTO")
    rects2 = ax.bar(x - (width * 1.0), lto, width, label="LTO_100")

    rects3 = ax.bar(x + (width * 0.0), cfi, width, label="CFI")

    rects3 = ax.bar(x + (width * 1.0), cfi, width, label="CFI_100")
    rects4 = ax.bar(x + (width * 2.0), scs, width, label="SCS")
    rects4 = ax.bar(x + (width * 3.0), scs, width, label="SCS_100")

    ax.set_ylabel("Bandwidth Gb/s")
    ax.set_xlabel("BM_sendVec_binder payload size (bytes)")
    ax.set_title("binder throughput")
    ax.set_xticks(x)
    ax.set_xticklabels(labels)
    ax.set_yscale("log", basey=2)
    ax.legend()
    plt.xticks(rotation=90)

    fig.savefig("binder_throughput_results.pdf", bbox_inches="tight")


def plot_binder_latency():
    num_pairs = 3
    results = {}
    for target in targets:
        filename = "binder_latency_{}.txt".format(target)
        with open(filename, "r") as f:
            data = json.load(f)

        # gather all pairs
        other_ms = 0.0
        fifo_ms = 0.0
        for i in range(num_pairs):
            k = "P{}".format(i)
            other_ms_avg = data[k]["other_ms"]["avg"]
            fifo_ms_avg = data[k]["fifo_ms"]["avg"]
            other_ms += other_ms_avg
            fifo_ms += fifo_ms_avg

        other_ms /= num_pairs
        fifo_ms /= num_pairs

        results[target] = (other_ms, fifo_ms)

    labels = [ "other_ms", "fifo_ms" ]
    clang = results["clang"]
    lto = results["lto"]
    lto100 = results["lto100"]
    cfi = results["cfi"]
    cfi100 = results["cfi100"]
    scs = results["scs"]
    scs100 = results["scs100"]

    x = np.arange(len(labels))
    width = 0.12

    fig, ax = plt.subplots()
    rects1 = ax.bar(x - (width * 3.0), clang, width, label="clang")
    rects2 = ax.bar(x - (width * 2.0), lto, width, label="LTO")
    rects3 = ax.bar(x - (width * 1.0), lto100, width, label="LTO_100")
    rects4 = ax.bar(x + (width * 0.0), cfi, width, label="CFI")
    rects5 = ax.bar(x + (width * 1.0), cfi100, width, label="CFI_100")
    rects6 = ax.bar(x + (width * 2.0), scs, width, label="SCS")
    rects7 = ax.bar(x + (width * 3.0), scs100, width, label="SCS_100")
    ax.set_ylabel("ms")
    ax.set_title("binder latency")
    ax.set_xticks(x)
    ax.set_xticklabels(labels)
    ax.legend()
    plt.xticks(rotation=90)

    fig.savefig("binder_latency_results.pdf", bbox_inches="tight")


def plot_hwbinder_latency():
    # hwbinder latency results have an ALL key that can be used to
    # get the average for all runs
    results = {}
    for target in targets:
        filename = "hwbinder_latency_{}.txt".format(target)
        with open(filename, "r") as f:
            data = json.load(f)

        # gather all pairs
        other_ms = data["ALL"]["other_ms"]["avg"]
        fifo_ms = data["ALL"]["fifo_ms"]["avg"]

        results[target] = (other_ms, fifo_ms)

    labels = [ "other_ms", "fifo_ms" ]
    clang = results["clang"][:2]
    lto = results["lto"][:2]
    lto100 = results["lto100"][:2]
    cfi = results["cfi"][:2]
    cfi100 = results["cfi100"][:2]
    scs = results["scs"][:2]
    scs100 = results["scs100"][:2]

    x = np.arange(len(labels))
    width = 0.12

    fig, ax = plt.subplots()
    rects1 = ax.bar(x - (width * 3.0), clang, width, label="clang")
    rects2 = ax.bar(x - (width * 2.0), lto, width, label="LTO")
    rects3 = ax.bar(x - (width * 1.0), lto100, width, label="LTO_100")
    rects4 = ax.bar(x + (width * 0.0), cfi, width, label="CFI")
    rects5 = ax.bar(x + (width * 1.0), cfi100, width, label="CFI_100")
    rects6 = ax.bar(x + (width * 2.0), scs, width, label="SCS")
    rects7 = ax.bar(x + (width * 3.0), scs100, width, label="SCS_100")
    ax.set_ylabel("ms")
    ax.set_title("hwbinder latency")
    ax.set_xticks(x)
    ax.set_xticklabels(labels)
    ax.legend()
    plt.xticks(rotation=90)

    fig.savefig("hwbinder_latency_results.pdf", bbox_inches="tight")


def plot_hwbinder_througput():
    results = {}
    for target in targets:
        results[target] = {}
        filename = "hwbinder_throughput_{}.txt".format(target)
        with open(filename, "r") as f:
            data = json.load(f)
        benchmarks = data["benchmarks"]
        for bench in benchmarks:
            name = bench["name"]
            if "_mean" not in name:
                continue
            size = int(name.split("/")[1].split("_")[0])
            iterations = bench["iterations"]
            time = bench["real_time"]

            # b/ns
            # for an approximate result, multiply the data
            # transfer rate value by 1.074
            throughput = (size * iterations) / time
            print(name)
            print("({} * {}) / {}".format(size, iterations, time))
            print("throughput: {:.2f} b/ns".format(throughput))
            print("throughput: {:.2f} Gb/s".format(throughput * 1.074))
            if not name in results[target]:
                results[target][name] = []
            results[target][name].append(throughput)

    # calculate average of results
    avg_results = {}
    for target in targets:
        avg_results[target] = {}

    for key, value in results.items():
        # calculate average
        res = results[key]

        for bench_name, bench_val in value.items():
            avg = sum(bench_val) / len(bench_val)
            value[bench_name] = avg

    clang = results["clang"].values()
    lto = results["lto"].values()
    lto100 = results["lto100"].values()
    cfi = results["cfi"].values()
    cfi100 = results["cfi100"].values()
    scs = results["scs"].values()
    scs100 = results["scs100"].values()

    labels = [ x.split("/")[1].split("_")[0] for x in results["clang"].keys() ]

    x = np.arange(len(labels))
    width = 0.12

    fig, ax = plt.subplots()

    rects1 = ax.bar(x - (width * 3.0), clang, width, label="clang")
    rects2 = ax.bar(x - (width * 2.0), lto, width, label="LTO")
    rects3 = ax.bar(x - (width * 1.0), lto100, width, label="LTO_100")
    rects4 = ax.bar(x + (width * 0.0), cfi, width, label="CFI")
    rects5 = ax.bar(x + (width * 1.0), cfi100, width, label="CFI_100")
    rects6 = ax.bar(x + (width * 2.0), scs, width, label="SCS")
    rects7 = ax.bar(x + (width * 3.0), scs100, width, label="SCS_100")

    ax.set_ylabel("Bandwidth Gb/s")
    ax.set_title("hwbinder throughput")
    ax.set_xticks(x)
    ax.set_xlabel("BM_sendVec_binderize")
    ax.set_xticklabels(labels)
    ax.set_yscale("log", basey=2)
    ax.legend()
    plt.xticks(rotation=90)

    fig.savefig("hwbinder_throughput_results.pdf", bbox_inches="tight")

```

```python

def main():
    plot_binder_throughput()
    plot_binder_latency()
    plot_hwbinder_latency()
    plot_hwbinder_througput()


if __name__ == "__main__":
    main()
```

A.7 WireGuard benchmarking

The script used for benchmarking WireGuard bandwidth is based on the same used in the Phoronix Test Suite. It is made by the WireGuard author, Jason A. Donenfeld. The script was downloaded from openbenchmarking.org, where many benchmarking suites for the Phoronix Test Suite are located. The url used was `http://www.phoronix-test-suite.com/benchmark-files/wireguard-for-pts-1.tar.xz`.

```bash
#!/bin/bash
# SPDX-License-Identifier: GPL-2.0
#
# Copyright (C) 2015-2020 Jason A. Donenfeld <Jason@zx2c4.com>.
# All Rights Reserved.
#
# NOTE: the original comment about topology is removed since LaTeX is crap at
#       handling UTF-8
set -e

exec 3>&1
export LANG=C
netns0="wg-test-$$-0"
netns1="wg-test-$$-1"
netns2="wg-test-$$-2"
pretty() { echo -e "\x1b[32m\x1b[1m[+] ${1:+NS$1: }${2}\x1b[0m" >&3; }
pp() { pretty "" "$*"; "$@"; }
maybe_exec() { if [[ $BASHPID -eq $$ ]]; then "$@"; else exec "$@"; fi; }
n0() { pretty 0 "$*"; maybe_exec ip netns exec $netns0 "$@"; }
n1() { pretty 1 "$*"; maybe_exec ip netns exec $netns1 "$@"; }
n2() { pretty 2 "$*"; maybe_exec ip netns exec $netns2 "$@"; }
ip0() { pretty 0 "ip $*"; ip -n $netns0 "$@"; }
ip1() { pretty 1 "ip $*"; ip -n $netns1 "$@"; }
ip2() { pretty 2 "ip $*"; ip -n $netns2 "$@"; }
waitiperf() { pretty "${1//*-}" "wait for iperf:5201 pid $2";
        while [[ $(ss -N "$1" -tlpH 'sport = 5201') != *\"iperf3\",pid=$2,fd=* ]];
        do sleep 0.1; done;
}
ping6="ping6"
```

```
type $ping6 >/dev/null 2>&1 || ping6="ping -6"

cleanup() {
	set +e
	exec 2>/dev/null
	ip0 link del dev wg0
	ip1 link del dev wg0
	ip2 link del dev wg0
	local to_kill="$(ip netns pids $netns0) $(ip netns pids $netns1) \
		$(ip netns pids $netns2)"
	[[ -n $to_kill ]] && kill $to_kill
	pp ip netns del $netns1
	pp ip netns del $netns2
	pp ip netns del $netns0
	exit
}

trap cleanup EXIT

ip netns del $netns0 2>/dev/null || true
ip netns del $netns1 2>/dev/null || true
ip netns del $netns2 2>/dev/null || true
pp ip netns add $netns0
pp ip netns add $netns1
pp ip netns add $netns2
ip0 link set up dev lo

ip0 link add dev wg0 type wireguard
ip0 link set wg0 netns $netns1
ip0 link add dev wg0 type wireguard
ip0 link set wg0 netns $netns2
key1="$(pp wg genkey)"
key2="$(pp wg genkey)"
pub1="$(pp wg pubkey <<<"$key1")"
pub2="$(pp wg pubkey <<<"$key2")"

configure_peers() {
	ip1 addr add 192.168.241.1/24 dev wg0
	ip1 addr add fd00::1/24 dev wg0

	ip2 addr add 192.168.241.2/24 dev wg0
	ip2 addr add fd00::2/24 dev wg0

	n1 wg set wg0 \
		private-key <(echo "$key1") \
		listen-port 1 \
		peer "$pub2" \
			allowed-ips 192.168.241.2/32,fd00::2/128
	n2 wg set wg0 \
		private-key <(echo "$key2") \
                listen-port 2 \
                peer "$pub1" \
                        allowed-ips 192.168.241.1/32,fd00::1/128

        ip1 link set up dev wg0
        ip2 link set up dev wg0
}
configure_peers

tests() {
        # Ping over IPv4
        n2 ping -c 10 -f -W 1 192.168.241.1
        n1 ping -c 10 -f -W 1 192.168.241.2

        # Ping over IPv6
        n2 $ping6 -c 10 -f -W 1 fd00::1
        n1 $ping6 -c 10 -f -W 1 fd00::2

        # TCP over IPv4
        n2 iperf3 -s -1 -B 192.168.241.2 &
        waitiperf $netns2 $!
        n1 iperf3 -J --logfile tcp_ipv4_$1_ipv$2_outer_$(uname -r).json -Z \
                          -n 10G -c 192.168.241.2

        # TCP over IPv6
        n1 iperf3 -s -1 -B fd00::1 &
        waitiperf $netns1 $!
        n2 iperf3 -J --logfile tcp_ipv6_$1_ipv$2_outer_$(uname -r).json -Z \
                          -n 10G -c fd00::1

        # UDP over IPv4
        n1 iperf3 -s -1 -B 192.168.241.1 &
        waitiperf $netns1 $!
        n2 iperf3 -J --logfile udp_ipv4_$1_ipv$2_outer_$(uname -r).json -Z \
                          -n 10G -b 0 -u -c 192.168.241.1

        # UDP over IPv6
        n2 iperf3 -s -1 -B fd00::2 &
        waitiperf $netns2 $!
        n1 iperf3 -J --logfile udp_ipv6_$1_ipv$2_outer_$(uname -r).json -Z \
                          -n 10G -b 0 -u -c fd00::2
}

[[ $(ip1 link show dev wg0) =~ mtu\ ([0-9]+) ]] && \
        orig_mtu="${BASH_REMATCH[1]}"
big_mtu=$(( 34816 - 1500 + $orig_mtu ))

# Test using IPv4 as outer transport
n1 wg set wg0 peer "$pub2" endpoint 127.0.0.1:2
n2 wg set wg0 peer "$pub1" endpoint 127.0.0.1:1
```

```
tests "normal" "4"
ip1 link set wg0 mtu $big_mtu
ip2 link set wg0 mtu $big_mtu
tests "jumbo" "4"

ip1 link set wg0 mtu $orig_mtu
ip2 link set wg0 mtu $orig_mtu

# Test using IPv6 as outer transport
n1 wg set wg0 peer "$pub2" endpoint [::1]:2
n2 wg set wg0 peer "$pub1" endpoint [::1]:1
tests "normal" "6"
ip1 link set wg0 mtu $big_mtu
ip2 link set wg0 mtu $big_mtu
tests "jumbo" "6"
```

The modifications changes the output format to JSON and writes every benchmark to a separate file.

The following Python script was used to plot the WireGuard results:

```python
#!/usr/bin/env python3
import json
import os
import re
import sys
import matplotlib.pyplot as plt
import numpy as np
from collections import OrderedDict


targets = [ "gcc", "clang", "lto", "lto100", "cfi", "cfi100" ]

# Several tests are run:
# - normal MTU: v4 over v4, tcp
# - normal MTU: v4 over v6, tcp
# - normal MTU: v6 over v4, tcp
# - normal MTU: v6 over v6, tcp
# - normal MTU: v4 over v4, udp
# - normal MTU: v4 over v6, udp
# - normal MTU: v6 over v4, udp
# - normal MTU: v6 over v6, udp
# - jumbo MTU: v4 over v4, tcp
# - jumbo MTU: v4 over v6, tcp
# - jumbo MTU: v6 over v4, tcp
# - jumbo MTU: v6 over v6, tcp
# - jumbo MTU: v4 over v4, udp
# - jumbo MTU: v4 over v6, udp
# - jumbo MTU: v6 over v4, udp
# - jumbo MTU: v6 over v6, udp

# format:
```

```
# {tcp,udp}_ipv{4,6}_{normal,jumbo},ipv{4,6}_outer_uname.json


def generate_table(values, labels, bench_type, proto):
    targets = [ "gcc", "clang", "lto", "cfi", "lto100", "cfi100" ]
    out = """\\begin{table*}[!htbp]
    \\begin{adjustbox}{width=1\\textwidth}
    \\centering
    \\begin{tabular}{@{}lllllllllllllllll@{}}
        kernel"""
    for label in labels:
        out += " & {}".format(label.replace("_", "\\_"))
    out += " \\\\\n"
    out += "\\midrule\n"

    fixed_results = {}

    for target in targets:
        fixed_results[target] = []

    print(values)
    for i in range(len(labels)):
        res = []
        for target in targets:
            if proto == "udp":
                res.append(values[target][i])
            else:
                res.append(values[target][i][bench_type])

        # calculate the best result
        best = max(res)

        for i in range(len(res)):
            val = res[i]
            if val == best:
                res[i] = 0.0
            else:
                res[i] = (float(best - val) / val) * 100

        for i in range(len(targets)):
            target = targets[i]
            fixed_results[target].append(res[i])

    for target in targets:
        out += "\t\t{}".format(target)

        for result in fixed_results[target]:
            if result == 0.0:
                out += " & \\cellcolor{{green!50}}0.0".format(result)
            elif result > 5.0:
```

```python
                out += " & \\cellcolor{{red!50}}{:.2f}".format(result)
            else:
                out += " & \\cellcolor{{yellow!50}}{:.2f}".format(result)
        out += " \\\\\n"
    out += "\\bottomrule\n"
    out += "\\end{tabular}\n"
    out += "\\end{adjustbox}\n"
    out += "\\caption{{WireGuard {} {} bandwidth}}\n".format(proto, bench_type)
    out += "\\label{{table:wg_{}_{}_bandwidth_results}}\n".format(proto, bench_type)
    out += "\\end{table*}\n"

    print(out)

def main():
    files = os.listdir("wg_results")
    results = {}

    for target in targets:
        results[target] = {}

    for f in files:
        print("parsing file {}...".format(f))
        pattern = r"(tcp|udp)_ipv(4|6)_(normal|jumbo)_ipv(4|6)_outer_5.7.0-"
        pattern += r"rc2-([a-z0-9]+)\+.json"
        m = re.match(pattern, f)

        transport = m.group(1)
        proto = m.group(2)
        mtu = m.group(3)
        outer = m.group(4)
        name = m.group(5)

        res_name = "{}_ipv{}_{}_ipv{}".format(transport, proto, mtu, outer)
        if not results[name]:
            results[name]["tcp"] = {}
            results[name]["udp"] = {}

        with open("wg_results/" + f, "r") as jf:
            data = json.load(jf)

        end = data["end"]

        if "sum_sent" in end:
            results[name]["tcp"][res_name] = {}
            results[name]["tcp"][res_name]["received"] = \
                    end["sum_received"]["bits_per_second"]
            results[name]["tcp"][res_name]["sent"] = \
                    end["sum_sent"]["bits_per_second"]
        else:
            results[name]["udp"][res_name] = {}
```

```
                results[name]["udp"][res_name] = end["sum"]["bits_per_second"]

        # start with udp results
        gcc = results["gcc"]
        clang = results["clang"]
        lto = results["lto"]
        cfi = results["cfi"]
        lto100 = results["lto100"]
        cfi100 = results["cfi100"]

        width = 0.12

        gcc["udp"] = dict(sorted(gcc["udp"].items()))
        clang["udp"] = dict(sorted(clang["udp"].items()))
        lto["udp"] = dict(sorted(lto["udp"].items()))
        lto100["udp"] = dict(sorted(lto100["udp"].items()))
        cfi["udp"] = dict(sorted(cfi["udp"].items()))
        cfi100["udp"] = dict(sorted(cfi100["udp"].items()))

        gcc["tcp"] = dict(sorted(gcc["tcp"].items()))
        clang["tcp"] = dict(sorted(clang["tcp"].items()))
        lto["tcp"] = dict(sorted(lto["tcp"].items()))
        lto100["tcp"] = dict(sorted(lto100["tcp"].items()))
        cfi["tcp"] = dict(sorted(cfi["tcp"].items()))
        cfi100["tcp"] = dict(sorted(cfi100["tcp"].items()))

        labels = cfi100["udp"].keys()
        x = np.arange(len(labels))

        fig, ax = plt.subplots()

        rects1 = ax.bar(x - (width * 2.5), gcc["udp"].values(),
                        width, label="gcc")
        rects2 = ax.bar(x - (width * 1.5), clang["udp"].values(),
                        width, label="clang")
        rects3 = ax.bar(x - (width * 0.5), lto["udp"].values(),
                        width, label="lto")
        rects4 = ax.bar(x + (width * 0.5), lto100["udp"].values(),
                        width, label="lto100")
        rects5 = ax.bar(x + (width * 1.5), cfi["udp"].values(),
                        width, label="cfi")
        rects6 = ax.bar(x + (width * 2.5), cfi100["udp"].values(),
                        width, label="cfi100")

        ax.set_ylabel("Throughput")
        ax.set_title("WireGuard")
        ax.set_xticks(x)
        ax.set_xticklabels(labels)
        ax.legend()
        plt.xticks(rotation=90)
```

```
    fig.savefig("wireguard_results_udp.pdf", bbox_inches="tight")

    udp_values = { "gcc": list(gcc["udp"].values()),
                   "clang": list(clang["udp"].values()),
                   "lto": list(lto["udp"].values()),
                   "lto100": list(lto100["udp"].values()),
                   "cfi": list(cfi["udp"].values()),
                   "cfi100": list(cfi100["udp"].values()) }

    generate_table(udp_values, labels, "send", "udp")

    # now tcp receive results
    labels = results["cfi100"]["tcp"].keys()
    x = np.arange(len(labels))

    width = 0.12
    fig, ax = plt.subplots()

    rects1 = ax.bar(x - (width * 2.5), list(map(lambda x: x["received"],
                    gcc["tcp"].values())), width, label="gcc")
    rects2 = ax.bar(x - (width * 1.5), list(map(lambda x: x["received"],
                    clang["tcp"].values())), width, label="clang")
    rects3 = ax.bar(x - (width * 0.5), list(map(lambda x: x["received"],
                    lto["tcp"].values())), width, label="lto")
    rects4 = ax.bar(x + (width * 0.5), list(map(lambda x: x["received"],
                    lto100["tcp"].values())), width, label="lto100")
    rects5 = ax.bar(x + (width * 1.5), list(map(lambda x: x["received"],
                    cfi["tcp"].values())), width, label="cfi")
    rects6 = ax.bar(x + (width * 2.5), list(map(lambda x: x["received"],
                    cfi100["tcp"].values())), width, label="cfi100")

    ax.set_ylabel("Throughput")
    ax.set_title("WireGuard TCP receive")
    ax.set_xticks(x)
    ax.set_xticklabels(labels)
    ax.legend()
    plt.xticks(rotation=90)
    fig.savefig("wireguard_results_tcp_recv.pdf", bbox_inches="tight")
    tcp_values = { "gcc": list(gcc["tcp"].values()),
            "clang": list(clang["tcp"].values()),
            "lto": list(lto["tcp"].values()),
            "lto100": list(lto100["tcp"].values()),
            "cfi": list(cfi["tcp"].values()),
            "cfi100": list(cfi100["tcp"].values()) }

    generate_table(tcp_values, labels, "received", "tcp")

    # now tcp sent results
    labels = results["cfi100"]["tcp"].keys()
    x = np.arange(len(labels))
```

```python

    width = 0.12
    fig, ax = plt.subplots()

    rects1 = ax.bar(x - (width * 2.5), list(map(lambda x: x["sent"],
                    gcc["tcp"].values())), width, label="gcc")
    rects2 = ax.bar(x - (width * 1.5), list(map(lambda x: x["sent"],
                    clang["tcp"].values())), width, label="clang")
    rects3 = ax.bar(x - (width * 0.5), list(map(lambda x: x["sent"],
                    lto["tcp"].values())), width, label="lto")
    rects4 = ax.bar(x + (width * 0.5), list(map(lambda x: x["sent"],
                    lto100["tcp"].values())), width, label="lto100")
    rects5 = ax.bar(x + (width * 1.5), list(map(lambda x: x["sent"],
                    cfi["tcp"].values())), width, label="cfi")
    rects6 = ax.bar(x + (width * 2.5), list(map(lambda x: x["sent"],
                    cfi100["tcp"].values())), width, label="cfi100")

    ax.set_ylabel("Throughput")
    ax.set_title("WireGuard TCP send")
    ax.set_xticks(x)
    ax.set_xticklabels(labels)
    ax.legend()
    plt.xticks(rotation=90)
    fig.savefig("wireguard_results_tcp_send.pdf", bbox_inches="tight")
    generate_table(tcp_values, labels, "sent", "tcp")


if __name__ == "__main__":
    main()
```

Appendix B

Code

B.1 CFI failure fixes

The following patch fixes a CFI error because of a parameter mismatch in the callback function passed to `is_mmconf_reserved()` in `pci_mmcfg_check_reserved()`. To avoid the CFI error a new callback friendly variant of `e820__mapped_all()` is introduced which has the correct function signature.

```
diff --git a/arch/x86/include/asm/e820/api.h b/arch/x86/include/asm/e820/api.h
index e8f58ddd06d9..a3de25ee4f3d 100644
--- a/arch/x86/include/asm/e820/api.h
+++ b/arch/x86/include/asm/e820/api.h
@@ -13,6 +13,7 @@ extern unsigned long pci_mem_start;
 extern bool e820__mapped_raw_any(u64 start, u64 end, enum e820_type type);
 extern bool e820__mapped_any(u64 start, u64 end, enum e820_type type);
 extern bool e820__mapped_all(u64 start, u64 end, enum e820_type type);
+extern bool e820__mapped_all_cb(u64 start, u64 end, unsigned type);

 extern void e820__range_add   (u64 start, u64 size, enum e820_type type);
 extern u64  e820__range_update(u64 start, u64 size, enum e820_type old_type, enum e820_type new_type);
diff --git a/arch/x86/kernel/e820.c b/arch/x86/kernel/e820.c
index c5399e80c59c..5f6ceb7d1e46 100644
--- a/arch/x86/kernel/e820.c
+++ b/arch/x86/kernel/e820.c
@@ -103,6 +103,7 @@ bool e820__mapped_any(u64 start, u64 end, enum e820_type type)
 }
 EXPORT_SYMBOL_GPL(e820__mapped_any);

+
 /*
  * This function checks if the entire <start,end> range is mapped with 'type'.
  *
@@ -150,6 +151,11 @@ bool __init e820__mapped_all(u64 start, u64 end, enum e820_type type)
        return __e820__mapped_all(start, end, type);
 }

+bool __init e820__mapped_all_cb(u64 start, u64 end, unsigned type)
+{
+       return e820__mapped_all(start, end, (enum e820_type)type);
+}
+
 /*
  * This function returns the type associated with the range <start,end>.
  */
```

```
diff --git a/arch/x86/pci/mmconfig-shared.c b/arch/x86/pci/mmconfig-shared.c
index 6fa42e9c4e6f..441d91c888a0 100644
--- a/arch/x86/pci/mmconfig-shared.c
+++ b/arch/x86/pci/mmconfig-shared.c
@@ -527,7 +527,7 @@ pci_mmcfg_check_reserved(struct device *dev, struct pci_mmcfg_region *cfg, int e
         /* Don't try to do this check unless configuration
            type 1 is available. how about type 2 ?*/
         if (raw_pci_ops)
-                return is_mmconf_reserved(e820__mapped_all, cfg, dev, 1);
+                return is_mmconf_reserved(e820__mapped_all_cb, cfg, dev, 1);

         return 0;
 }
```

B.2 Inline limit patch

For vanilla kernel.

```
diff --git a/Makefile b/Makefile
index 6444e5d024a6..a3da144939fc 100644
--- a/Makefile
+++ b/Makefile
@@ -882,7 +882,7 @@ endif
 CC_FLAGS_LTO_CLANG += -fvisibility=default

 # Limit inlining across translation units to reduce binary size
-LD_FLAGS_LTO_CLANG := -mllvm -import-instr-limit=5
+LD_FLAGS_LTO_CLANG := -mllvm -import-instr-limit=$(CONFIG_LTO_INLINE_LIMIT)
 KBUILD_LDFLAGS += $(LD_FLAGS_LTO_CLANG)

 KBUILD_LDS_MODULE += scripts/module-lto.lds
diff --git a/arch/Kconfig b/arch/Kconfig
index 98bc6e41821a..f3aafcc0836a 100644
--- a/arch/Kconfig
+++ b/arch/Kconfig
@@ -582,6 +582,17 @@ config THINLTO
         help
           Use ThinLTO to speed up Link Time Optimization.

+config LTO_INLINE_LIMIT
+        int "LTO inline limit"
+        depends on LTO_CLANG
+        default 5
+        help
+                This option controls the function size limit when inlining functions.
+                Increasing this number can lead to a high increase in binary size.
+                The number is the maximum number of instructions allowed for a function
+                to be inlined. So for the default value of 5, only functions with less
+                than 5 instructions will be inlined.
+
 choice
         prompt "Link-Time Optimization (LTO) (EXPERIMENTAL)"
         default LTO_NONE
```

For the Hikey 4.19 kernel:

```
diff --git a/Makefile b/Makefile
index 584aff800..787c2e345 100644
--- a/Makefile
+++ b/Makefile
@@ -822,7 +822,7 @@ endif
```

```diff
 lto-clang-flags += -fvisibility=default $(call cc-option, -fsplit-lto-unit)

 # Limit inlining across translation units to reduce binary size
-LD_FLAGS_LTO_CLANG := -mllvm -import-instr-limit=5
+LD_FLAGS_LTO_CLANG := -mllvm -import-instr-limit=$(CONFIG_LTO_INLINE_LIMIT)

 KBUILD_LDFLAGS += $(LD_FLAGS_LTO_CLANG)
 KBUILD_LDFLAGS_MODULE += $(LD_FLAGS_LTO_CLANG)
diff --git a/arch/Kconfig b/arch/Kconfig
index 6b6c82713..2c202e385 100644
--- a/arch/Kconfig
+++ b/arch/Kconfig
@@ -498,6 +498,17 @@ config THINLTO
         help
           Use ThinLTO to speed up Link Time Optimization.

+config LTO_INLINE_LIMIT
+        int "LTO inline limit"
+        depends on LTO_CLANG
+        default 5
+        help
+                This option controls the function size limit when inlining functions.
+                Increasing this number can lead to a high increase in binary size.
+                The number is the maximum number of instructions allowed for a function
+                to be inlined. So for the default value of 5, only functions with less
+                than 5 instructions will be inlined.
+
 choice
         prompt "Link-Time Optimization (LTO) (EXPERIMENTAL)"
         default LTO_NONE
```

Bibliography

[1] Martín Abadi et al. “Control-flow integrity principles, implementations, and applications.” In: *ACM Transactions on Information and System Security (TISSEC)* 13.1 (2009), pp. 1–40.

[2] *Active kernel releases.* `https://www.kernel.org/category/releases.html`. [Online; accessed 30-January-2020]. 2018.

[3] One Aleph. “Smashing the stack for fun and profit.” In: *Phrack 49* (1996).

[4] Android. *Building Kernels.* `https://source.android.com/setup/build/building-kernels`. [Online; accessed 06-May-2020].

[5] Android. *Performance Testing.* `https://source.android.com/compatibility/vts/performance`. [Online; accessed 02-May-2020].

[6] Android. *Using Binder IPC.* `https://source.android.com/devices/architecture/hidl/binder-ipc`. [Online; accessed 06-May-2020].

[7] ARM. *ARM® Cortex®-A Series - Programmer’s Guide for ARMv8-A.* English. Mar. 24, 2015.

[8] Roberto Avanzi. “The QARMA block cipher family. Almost MDS matrices over rings with zero divisors, nearly symmetric even-mansour constructions with non-involutory central rounds, and search heuristics for low-latency s-boxes.” In: *IACR Transactions on Symmetric Cryptology* (2017), pp. 4–44.

[9] John Aycock. “A brief history of just-in-time.” In: *ACM Computing Surveys (CSUR)* 35.2 (2003), pp. 97–113.

[10] Brandon Azad. *Examining Pointer Authentication on the iPhone XS.* `https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html`. 2019.

[11] Fabrice Bellard. “QEMU, a fast and portable dynamic translator.” In: *USENIX Annual Technical Conference, FREENIX Track.* 2005, pp. 41–46.

[12] Sandeep Bhatkar, Daniel C DuVarney, and Ron Sekar. “Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits.” In: *USENIX Security Symposium.* Vol. 12. 2. 2003, pp. 291–301.

[13] Joe Bialek et al. “Security analysis of memory tagging.” In: (2020).

[14] Chong Xu Bing Sun Jin Liu. *How to Survive the Hardware-assisted Controlflow Integrity Enforcement.* https://i.blackhat.com/asia-19/Thu-March-28/bh-asia-Sun-How-to-Survive-the-Hardware-Assisted-Control-Flow-Integrity-Enforcement.pdf.

[15] Tyler Bletsch et al. "Jump-oriented programming: a new class of code-reuse attack." In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security.* 2011, pp. 30–40.

[16] Daniel Pierre Bovet. *Implementing virtual system calls.* https://lwn.net/Articles/615809/. [Online; accessed 05-February-2020].

[17] Canonical. *Ubuntu.* https://ubuntu.com/.

[18] *ClangBuiltLinux.* https://clangbuiltlinux.github.io/. [Online; accessed 22-January-2020].

[19] *ClangBuiltLinux GitHub.* https://github.com/ClangBuiltLinux/linux/. [Online; accessed 22-January-2020].

[20] Kees Cook. *experimenting with Clang CFI on upstream Linux.* https://outflux.net/blog/archives/2019/11/20/experimenting-with-clang-cfi-on-upstream-linux/. [Online; accessed 23-April-2020]. 2019.

[21] Jonathan Corbet. *Kernel support for control-flow enforcement.* https://lwn.net/Articles/758245/.

[22] Jonathan Corbet and Greg Kroah-Hartman. *Linux Kernel Development Report.* https://www.linuxfoundation.org/2017-linux-kernel-report-landing-page/. [Online; 28-April-2019]. 2017.

[23] Crispan Cowan et al. "Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks." In: *USENIX security symposium.* Vol. 98. San Antonio, TX. 1998, pp. 63–78.

[24] *Debian.* https://www.debian.org/.

[25] Solar Designer. "Getting around non-executable stack (and fix)." In: *Bugtraq* (1997).

[26] Jason A. Donenfeld. "WireGuard: Next Generation Kernel Network Tunnel." In: *Proceedings of the 2017 Network and Distributed System Security Symposium.* Document ID: 4846ada1492f5d92198df154f48c3d54205657bc. San Diego, CA, 2017. ISBN: 1-891562-46-0. URL: https://www.wireguard.com/papers/wireguard.pdf.

[27] Jake Edge. *Namespaces in operation, part 7: Network namespaces.* https://lwn.net/Articles/580893/. [Online; accessed 22-April-2020]. Jan. 22, 2014.

[28] Ted Eisenberg et al. "The Cornell commission: on Morris and the worm." In: *Communications of the ACM* 32.6 (1989), pp. 706–709.

[29] glibc. *Locales in GLIBC.* https://sourceware.org/glibc/wiki/Locales. [Online; accessed 05-February-2020].

[30] *GNU.* https://www.gnu.org/.

[GPL] *GNU General Public License.* Version 2. Free Software Foundation, June 1991. URL: https://www.gnu.org/licenses/old-licenses/gpl-2.0.en.html.

[31] Google. *Android.* https://www.android.com/.

[32] Istvan Haller et al. “Typesan: Practical type confusion detection.” In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security.* 2016, pp. 517–528.

[33] Hex-Rays. *IDA Pro.* https://www.hex-rays.com/products/ida/index.shtml.

[34] Hong Hu et al. “Data-oriented programming: On the expressiveness of non-control data attacks.” In: *2016 IEEE Symposium on Security and Privacy (SP).* IEEE. 2016, pp. 969–986.

[35] Docker Inc. *Docker.* https://www.docker.com/.

[36] Intel. *Control-flow Enforcement Technology Specification.* https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf. [Online; accessed 31-January-2020]. May 2019.

[37] Vasileios P Kemerlis, Georgios Portokalidis, and Angelos D Keromytis. “kGuard: lightweight kernel protection against return-to-user attacks.” In: *21st {USENIX} Security Symposium ({USENIX} Security 12).* 2012, pp. 459–474.

[38] Michael Kerrisk. *Namespaces in operation, part 1: namespaces overview.* https://lwn.net/Articles/531114/. [Online; accessed 22-April-2020]. Jan. 4, 2013.

[39] Michael Kerrisk. *Namespaces in operation, part 2: the namespaces API.* https://lwn.net/Articles/531381/. [Online; accessed 22-April-2020]. Jan. 8, 2013.

[40] Michael Kerrisk. *Namespaces in operation, part 3: PID namespaces.* https://lwn.net/Articles/531419/. [Online; accessed 22-April-2020]. Jan. 16, 2013.

[41] Michael Kerrisk. *Namespaces in operation, part 4: more on PID namespaces.* https://lwn.net/Articles/532748/. [Online; accessed 22-April-2020]. Jan. 23, 2013.

[42] Michael Kerrisk. *Namespaces in operation, part 5: User namespaces.* https://lwn.net/Articles/532593/. [Online; accessed 22-April-2020]. Feb. 27, 2013.

[43] Michael Kerrisk. *Namespaces in operation, part 6: more on user namespaces.* https://lwn.net/Articles/540087/. [Online; accessed 22-April-2020]. Mar. 6, 2013.

[44] Chris Lattner. “LLVM: An Infrastructure for Multi-Stage Optimization.” *See* http://llvm.cs.uiuc.edu. MA thesis. Urbana, IL: Computer Science Dept., University of Illinois at Urbana-Champaign, Dec. 2002.

[45] *Linux.* https://www.kernel.org.

[46] Moritz Lipp et al. “Meltdown.” In: *arXiv preprint arXiv:1801.01207* (2018).

[47] LLVM. *LLVM Link Time Optimization: Design and Implementation.* https://llvm.org/docs/LinkTimeOptimization.html. [Online; accessed 25-May-2020].

[48] LLVM. *ShadowCallStack. Clang 11 documentation.* https://clang.llvm.org/docs/ShadowCallStack.html. [Online; accessed 31-January-2020].

[49] LLVM. *THINLTO.* https://llvm.org/docs/LinkTimeOptimization.html. [Online; accessed 25-May-2020].

[50] LLVM. *UndefinedBehaviorSanitizer.* https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html. [Online; accessed 25-May-2020].

[51] *LLVM.org.* https://llvm.org/. [Online; accessed 21-January-2020].

[52] H.J. Lu. *Control-flow Enforcement Technology.* https://www.linuxplumbersconf.org/event/2/contributions/147/attachments/72/83/CET-LPC-2018.pdf. Nov. 2018.

[53] Linux Programmer's Manual. *elf - format of Executable and Linking Format (ELF) files.* http://man7.org/linux/man-pages/man5/elf.5.html. [Online; accessed 05-February-2020].

[54] Linux Programmer's Manual. *ld.so, ld-linux.so - dynamic linker/loader.* http://man7.org/linux/man-pages/man8/ld-linux.so.8.html. [Online; accessed 05-February-2020].

[55] Linux Programmer's Manual. *pthreads - POSIX threads.* http://man7.org/linux/man-pages/man7/pthreads.7.html. [Online; accessed 10-April-2020].

[56] Matthew Tippett Michael Larabel. *Phoronix Test Suite.* http://phoronix-test-suite.com/. [Online; accessed 31-January-2020].

[57] Microsoft. *Data Execution Prevention.* https://docs.microsoft.com/en-us/windows/win32/memory/data-execution-prevention. [Online; accessed 22-May-2020]. 2018.

[58] João Moreira et al. "DROP THE ROP fine-grained control-flow integrity for the Linux kernel." In: *Black Hat Asia* (2017).

[59] Tim Newsham. *Format string attacks.* 2000.

[60] OpenMandriva. *OpenMandriva.* https://www.openmandriva.org/.

[61] OpenMandriva. *The best, until OpenMandriva does better: released OMLx 4.0.* https://www.openmandriva.org/en/news/article/the-best-until-openmandriva-does-better-released-omlx-4-0. [Online; accessed 06-June-2020].

[62] Jan Magnus Granberg Opsahl. "Open-source virtualization. Functionality and performance of Qemu/KVM, Xen, Libvirt and VirtualBox." MA thesis. University of Oslo, Department of Informatics, 2013.

[63] Sebastian Österlund et al. "kMVX: Detecting Kernel Information Leaks with Multi-variant Execution." In: *ASPLOS.* Apr. 2019. URL: https://www.vusec.net/download/?t=papers/kmvx_asplos19.pdf.

[64] Bob Page. *A REPORT ON THE INTERNET WORM.* https://www.ee.ryerson.ca/~elf/hack/iworm.html. [Online; accessed 24-May-2020]. 1988.

[65] Captain Planet. “A eulogy for format strings.” In: *Phrack (Nov. 2010)* (2010).

[66] Chromium Project. *Memory safety.* `https://www.chromium.org/Home/chromium-security/memory-safety`. [Online; accessed 24-May-2020].

[67] Inc. Qualcomm Technologies. *Pointer Authentication on ARMv8.3 - Design and Analysis of the New Software Security Instructions.* `https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf`.

[68] Felix Schuster et al. “Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications.” In: *2015 IEEE Symposium on Security and Privacy.* IEEE. 2015, pp. 745–762.

[69] Hovav Shacham et al. “The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86).” In: *ACM conference on Computer and communications security.* New York, 2007, pp. 552–561.

[70] R. Shirey. *Internet Security Glossary, Version 2.* RFC 4949. RFC Editor, Aug. 2007, pp. 1–365. URL: `https://www.rfc-editor.org/rfc/rfc4949.txt`.

[71] syzkaller. *syzkaller - kernel fuzzer.* `https://github.com/google/syzkaller`. [Online; accessed 14-June-2020].

[72] PaX Team. *Address space layout randomization (ASLR).* `https://pax.grsecurity.net/docs/aslr.txt`.

[73] Caroline Tice et al. “Enforcing Forward-Edge Control-Flow Integrity in {GCC} & {LLVM}.” In: *23rd {USENIX} Security Symposium ({USENIX} Security 14).* 2014, pp. 941–955.

[74] Sami Tolvanen. *Control Flow Integrity in the Android kernel.* `https://android-developers.googleblog.com/2018/10/control-flow-integrity-in-android-kernel.html`. [Online; accessed 24-April-2020]. Oct. 10, 2018.

[75] Sami Tolvanen. *Linux.* `https://github.com/samitolvanen/linux`. [Online; accessed 23-April-2020].

[76] Sami Tolvanen. *Protecting against code reuse in the Linux kernel with Shadow Call Stack.* `https://security.googleblog.com/2019/10/protecting-against-code-reuse-in-linux_30.html`. [Online; accessed 21-May-2020]. Oct. 30, 2019.

[77] Minh Tran et al. “On the expressiveness of return-into-libc attacks.” In: *International Workshop on Recent Advances in Intrusion Detection.* Springer. 2011, pp. 121–141.

[78] Victor van der Veen et al. “The Dynamics of Innocent Flesh on the Bone: Code Reuse Ten Years Later.” In: *CCS.* Oct. 2017. URL: `http://vvdveen.com/publications/newton.pdf`.

[79] Arjan van de Ven. “New Security Enhancements in Red Hat Enterprise Linux v.3, update 3.” In: (2004).

[80] Rafal Wojtczuk. “The advanced return-into-lib (c) exploits: Pax case study.” In: *Phrack Magazine* (2001).

[81] yyu168. *linux cet*. `https://github.com/yyu168/linux_cet`.