Informatics

Mitigating Return Address Leaks with Software Diversity

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering und Internet Computing

eingereicht von

Felix Berlakovich, BSc
Matrikelnummer 00929233

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Dr.techn. Edgar Weippl

Wien, 12. Dezember 2019

Felix Berlakovich Edgar Weippl

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Informatics

Mitigating Return Address Leaks with Software Diversity

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering and Internet Computing

by

Felix Berlakovich, BSc
Registration Number 00929233

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Dr.techn. Edgar Weippl

Vienna, 12th December, 2019

Felix Berlakovich Edgar Weippl

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Felix Berlakovich, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 12. Dezember 2019

Felix Berlakovich

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Acknowledgements

I want to thank my advisor Univ.-Prof. Dr. Stefan Brunthaler for the many fruitful discussions that have led to this thesis, and for his ongoing advice and motivation. His gracious support and guidance set me onto the path towards a scientific career. Furthermore, I would like to thank Privatdoz. Mag.rer.soc.oec. Dipl.-Ing. Dr. techn. Edgar Weippl for his advice and supervision.

I also want to thank SBA Research, the company that has employed me during my master thesis, for allowing me to focus on my research without financial pressure.

My deepest gratitude goes to my parents Geraldine and Johann, as without their support I would not be where I am today. They have never ceased to encourage me to pursue my goals.

Finally, I would like to thank Christina Kerrigan for proof reading, for her care and for enduring me even at difficult times.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Ein erheblicher Teil heutiger Programme ist in speicher-unsicheren Sprachen wie C oder C++ programmiert. Speicher-unsichere Sprachen sind anfällig für Speicher-Korrumpierungen und als Folge davon für Kontrollflussübernahme (englisch „control-flow hijacking“). Trotz mehr als einem Jahrzehnt Forschung sind Kontrollflussübernahme-Attacken noch immer eine ernsthafte Sicherheitsbedrohung. Obwohl W⊕X und eine Randomisierung des Programmcode-Layouts solche Angriffe erschweren, sind sie kein Allheilmittel. Angreifer sind zu Angriffen übergegangen, bei denen bestehende Programmteile wiederverwendet werden (englisch „code-reuse attacks“) und haben begonnen geleckte Information zu verwenden, um die Randomisierung des Programmcode-Layouts auszuhebeln. Insbesondere indirekte Informationslecks (englisch „indirect information disclosure“) haben sich als schwierig zu bekämpfen erwiesen.

In dieser Arbeit analysieren wir öffentlich bekannte Attacken, die sich indirekte Informationslecks zunutze machen, sowie Verteidigungen dagegen. Basierend auf den Erkenntnissen der Analyse präsentieren wir Rad, eine neuartige Verteidigung, um indirekte Informationslecks zu verhindern, die durch Rücksprungadressen verursacht werden. Wir verwenden Sotwarediversität um das a priori Wissen im Kern von stack-basierten Angriffen, die sich indirekte Informationslecks zunutze machen, zu invalidieren. Außerdem diskutieren wir die Anwendbarkeit unserer Idee auf andere sensitive Zeiger wie C++ vtable Zeiger. Wir implementieren eine Prototyp-Implementierung von Rad auf Basis der LLVM Compiler-Infrastruktur. Abschließend evaluieren wir unseren Prototypen mithilfe der SPEC CPU2006 Benchmarksuite und führen eine gründliche theoretische Analyse der von Rad bereitgestellten Sicherheitsgarantien durch.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Abstract

A considerable part of today's software is written in memory unsafe languages, like C and C++. Memory unsafe languages are prone to memory corruptions and, as a result, to control-flow hijacking. Despite more than a decade of research, control-flow hijacking attacks are still a serious security threat. While W⊕X and code layout randomization have raised the bar for adversaries, they are not a panacea. Attackers have shifted to code-reuse attacks and have started to use information leaks to undermine the effects of code layout randomization. In particular, indirect information disclosure, a technique in which the attacker uses pointers to infer a randomized code layout, has proven to be difficult to counter.

In this thesis, we analyze publicly-known attacks which use indirect information disclosure as well as defenses against them. Based on the findings of the analysis, we present Rad, a novel defense which prevents indirect information disclosures enabled by return addresses. We use software diversity to invalidate the a-priori knowledge at the core of stack-based indirect information disclosure attacks. We also discuss how our idea can be applied to other sensitive pointers, such as C++ vtable pointers. We provide a prototype implementation of Rad based on the LLVM compiler infrastructure. Finally, we evaluate our prototype with the SPEC CPU2006 benchmark suite and give a thorough theoretical analysis of its security guarantees.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 1

Introduction

1.1 Motivation

Despite the advent of memory-safe programming languages like Java or Python, memory-unsafe languages like C and C++ are still widely used. According to the TIOBE Index [18], C is the second most widely-used programming language and C++ is in the fourth place. C and C++ are used to build software ranging from office applications to automotive software. In addition, the virtual machines underlying languages like Java or Python are also often written in C or C++. As a result, applications written in C or C++ are perceived as a target worthwhile attacking. As these applications are also used in safety-critical areas, successful attacks can have disastrous effects.

Memory-unsafe programming languages allow for arbitrary pointer arithmetic and access to the application's memory without bounds checking. While the lack of bounds checks can improve application's performance, programming errors can lead to memory-related vulnerabilities. Attackers can exploit these vulnerabilities and corrupt control-data to hijack the program's control-flow [Sha07; Che+10; Ble+11]. Although some defenses try to prevent this corruption of control-data, preventing all possible corruptions is hard. Hence, another class of defenses aims to contain the negative effects of control-flow hijacking. Software diversity belongs to this class.

Software diversity introduces artificial diversity into software to invalidate a-priori knowledge an attacker might have. This includes knowledge necessary to mount a successful control-flow hijacking attack. Attackers have responded to software diversity by leaking the diversified parts at runtime and, thus undermining the effects of software diversity. Introducing execute only memory [Bac+14; Wer+16] can prevent attackers from disclosing a diversified code layout. However, attackers might still use a technique called *indirect* memory disclosure. Indirect memory disclosure allows to infer the code layout from pointers found in readable parts of the memory.

Indirect memory disclosure can be countered with Code-Pointer Hiding (CPH) [Cra+15b]. With CPH, pointers in readable memory are redirected through an execute only trampoline. Unfortunately, CPH causes a significant performance impact by disrupting the CPUs branch prediction. Moreover CPH does not protect against code pointers being maliciously reused [Rud+17].

1.2 Problem statement

While software diversity can thwart a number of different attacks [Cra+15b], it relies on the secrecy of diversified parts of the software. Therefore, information leaks can undermine the benefits of software diversity. Indirect memory disclosure, in particular, can be hard to prevent and existing defenses have their own shortcomings.

This thesis aims to investigate new ways of defending against indirect memory disclosure attacks. We analyze existing attacks and defenses and propose a new technique called Return Address Decoys (Rad) to protect function return addresses from being leaked. We evaluate our technique by providing a proof of concept implementation for Linux based on the LLVM compiler suite. We also discuss the applicability of the idea to other areas where indirect memory disclosure poses a problem. Finally, we assess the the performance impact of our solution with a comprehensive set of benchmarks and perform a security evaluation.

1.3 Structure of the thesis

The rest of this thesis is structured as follows: Chapter 2 introduces relevant background information on operating system fundamentals, calling conventions, branch prediction and the development of increasingly sophisticated code-reuse attacks. Chapter 3 discusses existing defenses, that are similar to Rad. In Chapter 4 we explain the design choices, that have guided the implementation of Rad. Chapter 5 describes the cornerstones of our prototype implementation. In Chapter 6 we present the results of our evaluation of Rad. Finally, we discuss the results in Chapter 7 and conclude the thesis in Chapter 8.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 2

Background

In this chapter we give a basic overview of the topics required to understand our defense. We first introduce discuss the representation of processes in memory and explain calling conventions as well as branch prediction. We conclude the chapter with an overview of the development of increasingly sophisticated attacks and defenses that has led to the need for new solutions against memory disclosure.

2.1 Representation of processes in memory

When a user starts an application the operating system creates a process and loads the necessary data from permanent storage into memory. The following sections describe how a process receives an isolated view of memory with the help of virtual memory and how the process is represented in memory. More information on how operating systems represent processes can be found in textbooks such as *Operating Systems: Internals and Design Principles* or *Understanding the Linux Kernel* [Sta11; BC05].

2.1.1 Virtual memory

The memory management of CPUs provides a feature called *virtual memory*. Virtual memory introduces an indirection between the memory visible to a process and physical memory. This indirection allows the operating system to run processes in their own memory sandbox. Each process can virtually address the entire memory in a contiguous address space. Whenever a process tries to access a virtual memory address, the CPU transparently translates the virtual address into a physical address. The translation happens with the help of a data structure called *page table*.

The page table contains mappings between chunks of virtual memory and physical memory called *pages*. The operating system maintains a page table for each process. By giving each process a separate page table, every process has its own view of the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

system memory. The page table also enables the operating system to map virtual pages from different processes to the same physical page. This n:1 mapping enables sharing of physical memory pages, and, therefore, reduces the required amount of physical memory.

Entries in the page table have access permission flags. Using the permission flags, the operating system marks pages as readable, writable or non-executable. Furthermore, pages can be marked as privileged. Only kernel code can access such privileged pages. When a process violates access permissions (e.g., by writing to a read-only page) a fault occurs and the CPU notifies the operating system. Typically the offending process is terminated.

2.1.2 Process memory layout

The memory space addressable by a process is subdivided into two subspaces—the kernel space and the user space. The kernel space is used to map the kernel itself into the process's address space and is only accessible in privileged mode. Processes can execute system calls with special CPU instructions that switch between user space and kernel space. The code and data of a process resides in user space.

Under Linux the user space is further subdivided into the following sections:

- Stack

  The stack section or stack stores temporary function data. For example, the stack stores local variables and function parameters. The stack grows and shrinks contiguously[1], and data can only be added to or removed from the top. The data on the stack follows a last in, first out (LIFO) order and is temporary. Data pushed during the execution of a function is removed again when the function returns. That is, the lifetime of function data on the stack is bound to the lifetime of the function. A special register called *stack pointer register* always points to the top of the stack. The CPU provides specialized instructions for manipulating the stack and adjusting the stack pointer accordingly (e.g., `push` and `pop` instructions). Modern operating systems mark the stack as non-executable to prevent code injection attacks.

- Heap

  The heap is used for dynamic memory allocations. Memory allocated on the heap does not have to be contiguous. Applications can request memory from the heap at runtime, usually with the help of a runtime library. For example, memory allocated with the `malloc` library function is located on the heap. The lifetime of data on the heap is not bound to a single function. Data allocated by a function can stay on the heap after the function returns. Modern operating systems mark the heap as non-executable to prevent code injection attacks.

[1]The growth direction depends on the machine architecutre. On x86 the stack grows downwards, i.e., from high to low addresses.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- BSS

  The Block Started by Symbol (BSS) section contains static variables which are uninitialized at compile-time. The variables in this section are initialized with zero when the process is created. The BSS section is typically readable and writable, but not executable.

- Data

  The data section contains initialized static variables of the program. The data section contains a writable and a read-only part. The read-only part contains constant variables and other constant data (e.g. C++ vtables). The writable part hosts writable static variables. The data section is typically non-executable.

- Text

  The text section contains the executable code of the program and is typically not writable, but readable and executable. Marking the text section read-only allows the operating system to share memory between processes with the same code (e.g. multiple instances of the same application).

- Memory mapping section

  The memory mapping section is used to map code and data which are not part of the loaded program. For example, the dynamic linker uses this section to map dynamic libraries into the process address space. Memory mapped files are also mapped into this section. The access permissions of the memory mapping section depend on the data being mapped. For example, dynamic libraries are typically mapped readable and executable, whereas files are mapped non-executable.

Figure 2.1 gives an overview of the memory sections of a Linux process together with the typical access permissions. Note that the access permissions can be changed and also depend on the security features available in the running kernel.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

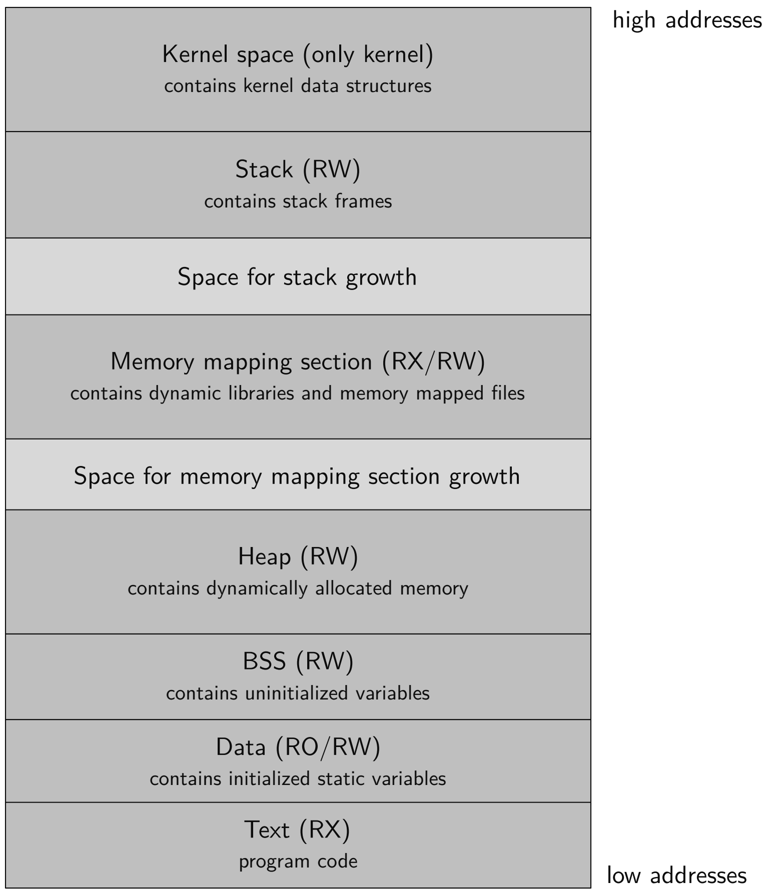

high addresses
Kernel space (only kernel)
contains kernel data structures
Stack (RW)
contains stack frames
Space for stack growth
Memory mapping section (RX/RW)
contains dynamic libraries and memory mapped files
Space for memory mapping section growth
Heap (RW)
contains dynamically allocated memory
BSS (RW)
contains uninitialized variables
Data (RO/RW)
contains initialized static variables
Text (RX)
program code
low addresses

Figure 2.1: Overview of the memory layout of a Linux process.

2.2 Calling conventions and stack layout

When a function calls another function, the caller and the callee must adhere to a certain interface. This interface is called *calling convention* and differs between machine architectures and operating systems. In particular, calling conventions define how parameters are passed, which registers a called function must preserve and how the stack frame setup is divided between caller and callee. Although most compilers and operating systems follow a specific calling convention, the calling convention is not enforced by hardware. The calling convention under Linux when running on a x86 machine architecture is called System V AMD64 Application Binary Interface (ABI). In the following section we describe how the stack is used in the System V AMD64 ABI calling convention. More details on calling conventions and stack layouts can be found in compiler textbooks, such as *Advanced Compiler Design and Implementation* [**Muchnick 1997**].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

2.2.1 System V AMD64 ABI and the stack

Calling a function pushes a data structure called *stack frame* onto the stack. A stack frame represents the runtime state of a single function invocation and contains control-flow information as well as data of the function. When the function returns, the corresponding stack frame is removed from the stack. Note that if a function is called multiple times, it has multiple stack frames on the stack.

Under Linux the stack grows downward from high memory addresses while the heap grows upward from low memory addresses[2]. The stack pointer always points to the bottom of the current stack frame (i.e., the top of the stack). A pointer called *base pointer* points to the beginning of the current stack frame. The base pointer is usually stored in the base pointer register.

A stack frame consists of the following elements:

- function arguments (optional)
- return address
- callee-saved registers (optional)
- saved base pointer (optional)
- local variables

A graphical representation of stack frames is shown in Figure 2.2. We explain the different parts of the stack frame by outlining the steps needed to call a function. The setup and destruction of the call frame can be divided into 4 steps:

1. Caller preparations

   Before transferring control to the callee, the caller begins the setup of the stack frame. Under certain conditions the caller preparations might be omitted (e.g., function which does not return without parameters and registers to be saved).

   a) Store the contents of all caller-saved registers which are in use. If the caller does not use certain caller-saved registers, it might skip saving them.
   b) Supply the arguments for the called function. The first 6 integer arguments and the first 8 floating point arguments are passed via the registers. Integer arguments use the registers `rdi`, `rsi`, `rdx`, `r8` and `r9` while floating point arguments use `xmm0-xmm7`. Every additional argument must be passed via the stack. The caller puts the arguments from right to left onto the stack which means, since the stack is LIFO, that the leftmost argument is closest to the top of the stack.

[2]Strictly speaking the stack growth direction is given by the machine architecture and the heap growth direction by the operating system. On x86 the stack grows downward.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

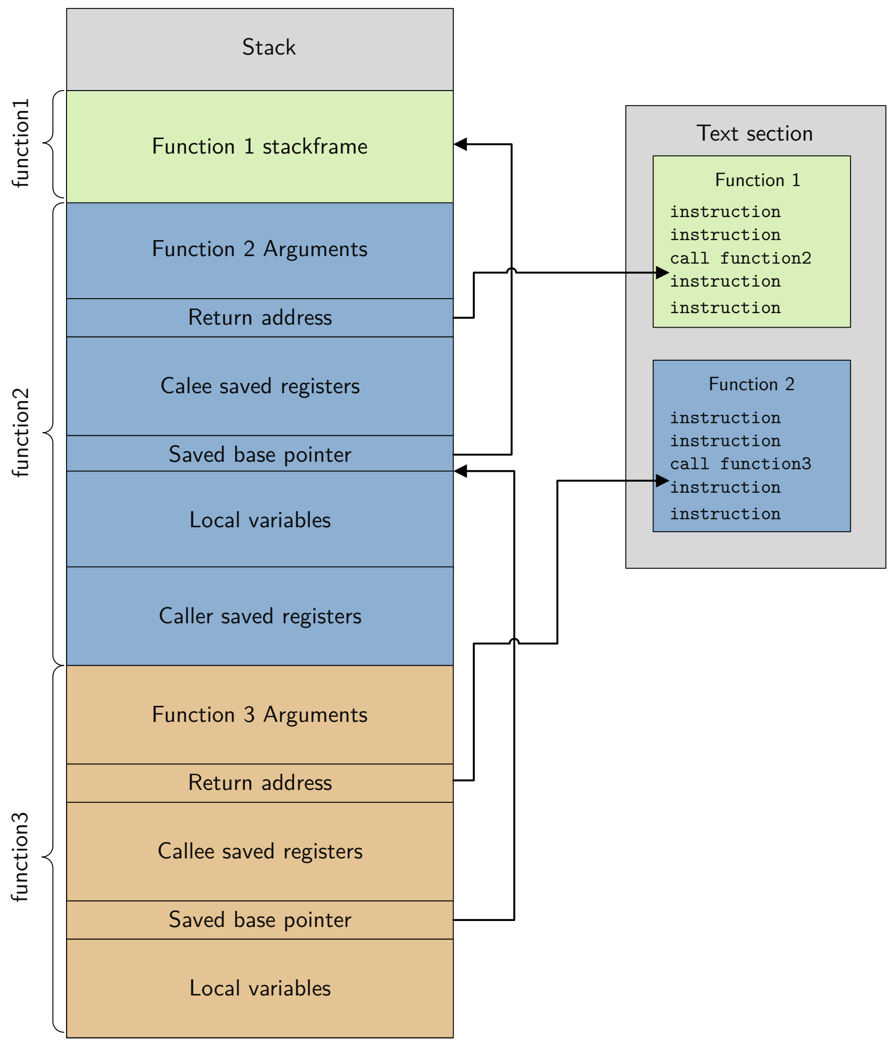


Figure 2.2: Stack frames on the stack.

c) Push the return address onto the stack. The return address determines where the execution continues after the function returns. The return address is a pointer to a location in the text section, usually the instruction after the call instruction. When calling a function with the x86 `call` instruction, the return address is automatically pushed onto the stack.

2. Function prologue

   The function prologue is a sequence of instructions at the beginning of a function, which performs the callee-side stack frame setup.

   a) Store the contents of the callee saved registers

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

   b) Store the base pointer of the calling function and let the base pointer register point to the beginning of the current stack frame. This enables addressing local variables and parameters relative to the base pointer register. Note that under certain conditions (e.g., if the stack frame size is known at compile time) local variables and parameters can be addressed relative to the stack pointer. This optimization frees the base pointer register for the use as a temporary register.
   c) Adjust the stack pointer to allocate stack space to hold local variables

3. Function epilogue

   The function epilogue reverses the actions of the function prologue and returns control to the caller

   a) Restore callee saved registers
   b) Restore the base pointer of the caller
   c) Adjust the stack pointer to deallocate the stack space for local variables
   d) Pop the return address from the stack and jump to the return address. When returning from a function with the `ret` instruction, the return address is automatically popped from the stack.

4. Restore the pre-call state

   After calling a function the caller restores the register state before the call

   a) Restore callee saved registers

2.3 CPU pipelines and branch prediction

The completion of an assembly instruction by the CPU is subdivided into multiple stages. The sequence of stages is called *pipeline*. An instruction completes, once it has moved through the pipeline (i.e., has completed every stage). For example, the pipeline of a Reduced Instruction Set Computer (RISC) CPU typically contains the following stages:

1. instruction fetch
2. instruction decode
3. execute
4. memory access
5. writeback

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

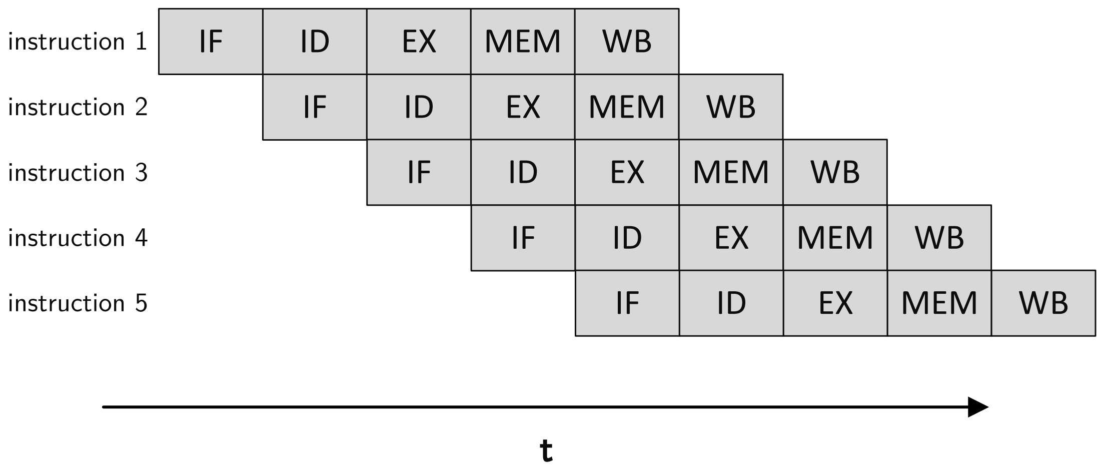


Figure 2.3: Pipeline with five stages: instruction fetch (IF); instruction decode (ID); execute (EX); memory access (MEM); writeback (WB).

The different stages in the pipeline are processed by different parts of the CPU, which can operate in parallel. If only a single instruction would move through the pipeline, 4 out of the 5 pipeline stages would stay empty. For that reason, the CPU tries to keep the pipeline filled by putting a new instruction in the pipeline whenever the previous instruction moves forward. For example, when the first instruction enters the „instruction decode“ stage, the next instruction can already enter the „instruction fetch“ stage. An example of a five-stage pipeline can be found in Figure 2.3.

To fetch a new instruction, the CPU must decide which instruction should be executed next. The simplest strategy is to fetch the instruction following the previous one in memory. This strategy works well for linear programs, i.e., programs without branches. However, if the program contains branch instructions (e.g., a `jmp` instruction), the instructions following a branch instruction should enter the pipeline only if the branch is not taken. If the branch is taken, the CPU should fill the pipeline with instructions at the branch target instead.

Branch targets often depend on values calculated by previous instructions. Even if the branch target does not depend on previous instructions, the branch instruction itself must pass the first three stages before the branch target is known. As a result, the required information to decide whether a branch is taken or not might not be available yet when the CPU needs to fetch the next instruction. Waiting for the information to become available would cause a *pipeline stall*, i.e. a situation in which stages in the pipeline cannot be filled. A pipeline stall leads to unused CPU resources and, therefore, negatively impacts performance. To avoid pipeline stalls, the CPU uses a component called *branch predictor*. The branch predictor tries to predict the branch target based on previous control-flow transfers. Mispredictions lead to wrong instructions entering the pipeline. In such a case the CPU must clear the pipeline to prevent incorrect results

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

which in turn also leads to a pipeline stall.

Modern CPUs use different branch prediction strategies for different purposes. For example, `ret` instructions are special branch instructions as they typically follow a `call` instruction. As a result, the branch predictor predicts the branch target of a `ret` instruction to be the return address of the current function. The return addresses are, in addition to the stack, kept in a special buffer called Return Stack Buffer (RSB), which is updated by `call` and `ret` instructions.

2.4 Exploitation and defense mechanisms

In this section we explain the development of ever more sophisticated arbitrary code execution attacks and defenses. Each subsection about an attack explains which of the following prerequisites is needed to mount the attack:

List of prerequisites

PRQ.1 a memory corruption to divert control flow

PRQ.2 a writable and executable buffer

PRQ.3 the address of injected code

PRQ.4 a writable buffer

PRQ.5 the addresses of useful functions or ROP gadgets

PRQ.6 a memory disclosure vulnerability to disclose the text section

PRQ.7 a memory disclosure vulnerability to disclose code addresses through pointers on the stack, heap or in the data section

PRQ.8 pointers at known locations on the stack, heap or in the data section

Each subsection about a defense explains which of these prerequisite defense tries to remove. Figure 2.4 gives a graphical overview of the development of attacks and defenses described in this section.

A common way to exploit a vulnerability in a program is *control-flow hijacking* [Sze+13]. Control-flow hijacking is a form of memory corruption attack, where the attacker tries to subvert the benign control flow of the program. For a control-flow hijacking attack the attacker must accomplish two tasks: 1) change the control flow of the program to 2) make the program perform actions of his choosing. The following sections describe these two tasks as well as proposed defenses in more detail.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

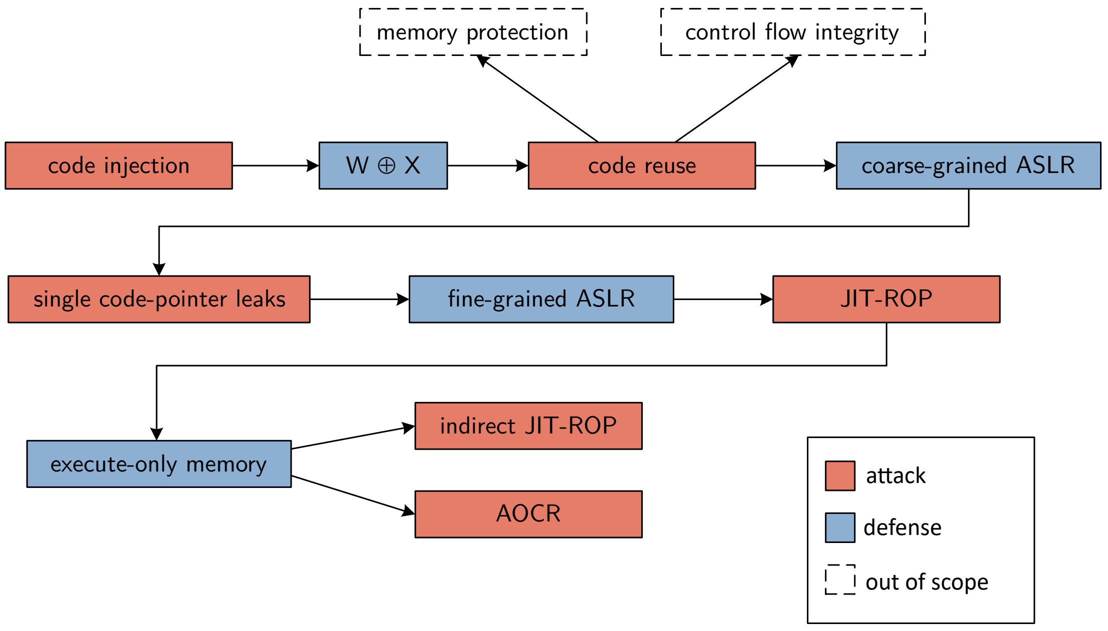


Figure 2.4: Overview of the development of attacks and defenses.

```c
void read_input() {
  char *buf[300];
  gets(buf);
}

int main() {
  read_input();
  return 0;
}
```

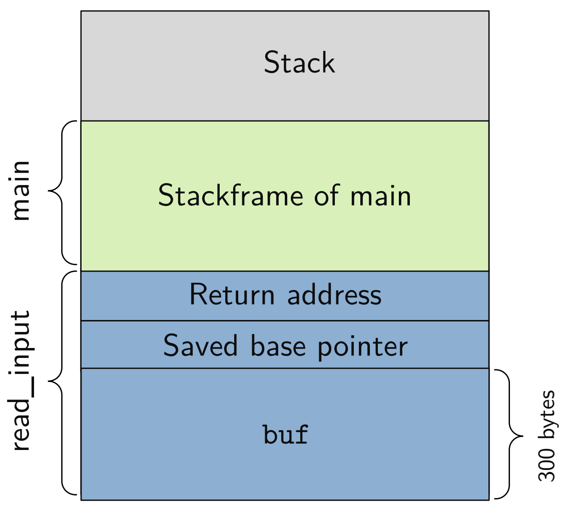
Listing 2.1: Simple buffer overflow vulnerability.

2.4.1 Diverting control flow

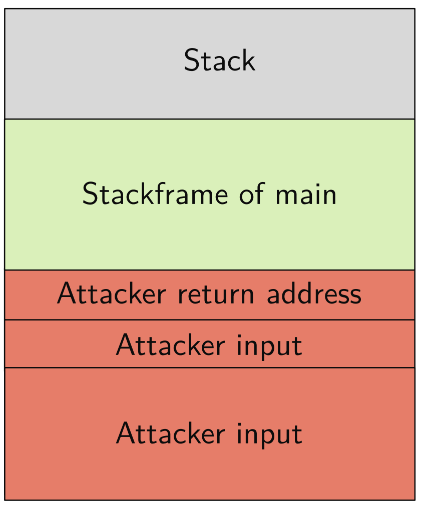
Attackers can divert the control flow of a program by using an out of bounds memory access to overwrite control-flow data. A prominent example of modified control-flow data are return addresses [One96]. For example, consider the C program in Listing 2.1:

After the prologue of `read_input` the stack looks like in Figure 2.5a. If an adversary enters more than 300 characters, `gets` starts to write beyond the space on the stack that was reserved for `buf`. First, the saved base pointer is overwritten and next the return address. The stack after the buffer overflow is shown in Figure 2.5b. Upon return of `read_input` the epilogue would now try to jump to the overwritten location instead of the original return address. Note that similar attacks exist with other types of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.



(a) Before the overflow.



(b) After the overflow.

Figure 2.5: Stack layout of the program in Listing 2.1 before and after the overflow.

control-flow data. For example, an attacker might overwrite subsequently dereferenced code pointers or to divert control flow [Ano01].

2.4.2 Performing attacker chosen actions

Once an attacker is able to overwrite control-flow data, she must decide where to divert the control flow to. The following sections explain different possibilities for control-flow diversion.

Code injection

Early versions of memory corruption attacks used the overflown buffer not only to overwrite control data, but also to inject the code to be executed. Such an attack is called *code injection attack*. In the example in Listing 2.1 the attacker has 300 bytes from the buffer plus the space used by the saved base pointer to store injected shell code onto the stack. After injecting code, the attacker overwrites the return address to point to the beginning of the buffer on the stack. Figure 2.6 shows the memory layout of the program in Listing 2.1 after the buffer overflow.

Prerequisites A successful code injection attack assumes the following prerequisites:

PRQ.1 a memory corruption to divert control flow;

PRQ.2 a writable and executable buffer to inject code;

PRQ.3 the address of injected code.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

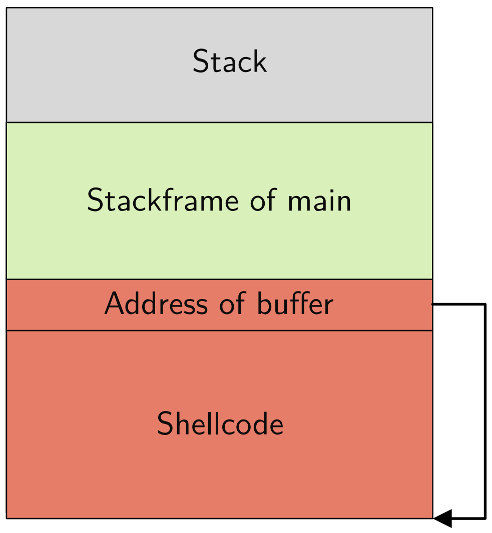


Figure 2.6: Memory layout of a program after a code injection overflow.

A defense called Write exclusive-or Execute (W⊕X), removes prerequisite **PRQ.2**. W⊕X ensures that writable memory areas are not executable at the same time. For example, with W⊕X in place, the stack section is only writable. Hence, a code injection attack with code on the stack like in Figure 2.6 is not possible anymore.

Reusing existing code

Attackers soon realized that injecting code is not strictly necessary to mount a successful attack. To circumvent W⊕X, an adversary can use code that is already present in the target process. Such attacks are called *code-reuse attacks*. The simplest form of code-reuse attacks reuses whole functions. For example, an attacker could reuse a function of the attacked program, as shown in Listing 2.2.

Instead of injecting new code into the buffer, the attacker overwrites the return address with the address of `launch_missiles`. If required, the attacker can use the buffer as a frame for the reused function to operate on. The text section, in which `launch_-missiles` resides, is guaranteed to be executable. Upon return of `read_input`, the program continues execution in the function `launch_missiles` instead of after the call to `read_input` (i.e., line 12). Figure 2.7 shows the memory layout of a process after a code-reuse attack.

Note that the reused function does not have to be part of the program itself. Another approach, called *return-to-libc*, reuses functions from the libraries used by the program. Such libraries are mapped into the the address space of the process at load time. The attack was initially demonstrated with the standard C library, `libc`, because `libc` is loaded in almost every Unix program. However, the return-to-libc attack also works with other libraries.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
void read_input() {
  char *buf[300];
  gets(buf);
}

void launch_missiles() {
  // do something harmful
}

int main() {
  read_input();
  return 0;
}
```

Listing 2.2: Code reuse vulnerability.

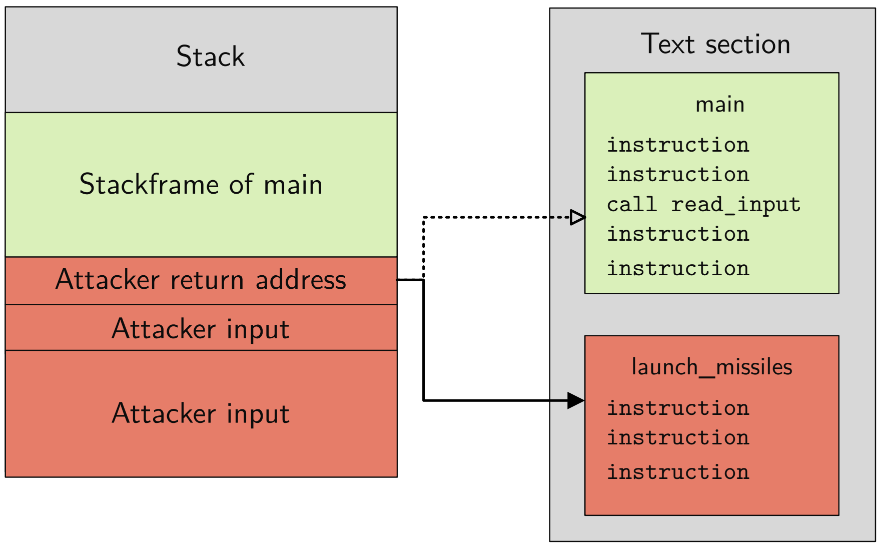


Figure 2.7: Memory layout of a program after a code-reuse attack.

Another form of code-reuse attack is Return-Oriented-Programming (ROP) [Sha07]. In a ROP attack, instead of reusing whole functions, an attacker strings together short sequences of assembly code which end in a `ret` instruction. Such a sequence is called *gadget*. Gadgets can already be part of the assembly code that is emitted by the compiler. However, the attacker can also use gadgets which result from executing unaligned code.

As an example for unaligned gadgets, consider the piece of assembly code in Listing 2.3. The code is taken from the UNIX program `ls` at address `0x6443`. When `ls` was compiled, the compiler emitted the following two instructions at address `0x6443`:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
6443:        0f 94 c2              sete   dl
6446:        08 c2                 or     al, dl
```

Listing 2.3: Original assembly code from the `ls` program.

```
6444:        94                    xchg eax, esp
6445:        c2 08 c2              ret 0xc208
```

Listing 2.4: Unaligned assembly code from the `ls` program.

1. the 3 byte long `sete dl` instruction consisting of the bytes `0f94c2`;
2. the 2 byte long `or al,dl` instruction consisting of the bytes `08c2`.

Reading the code with one byte offset though—starting at address `0x6444`—the byte sequence is interpreted as the two instructions shown in Listing 2.4:

1. the 1 byte long `xchg eax, esp` instruction consisting of the byte `94`;
2. the 3 byte long `ret 0xc208` instruction consisting of the bytes `c208c2`.

In contrast to the code in Listing 2.3, the code in Listing 2.4 constitutes a ROP gadget.

A ROP attack works by first identifying useful gadgets in an offline copy of the program. Next, the attacker writes the addresses of the gadgets in the target process onto the stack. Finally, the attacker diverts the control flow to the first gadget address on the stack. After the execution of the instructions in a gadget, the `ret` instruction at the end of the gadget pops the next address from the stack and transfers control to the next gadget.

In both, whole function reuse and ROP attacks, the attacker uses the writable buffer for the function or the ROP gadgets respectively. Instead of storing the gadget addresses on the stack, an attacker can also store the addresses into a buffer on the heap. In this case the first gadget needs to be a stack pivot gadget. A stack pivot gadget changes the stack pointer to point to a memory area other than the stack section. Subsequently, specialized CPU instructions that manipulate the stack (e.g `ret` or `pop`) treat the heap as stack. Moreover, a modified version of ROP uses different branch instructions (e.g., `jmp`) instead of `ret` instructions [Ble+11].

Prerequisites A successful code-reuse attack assumes the following prerequisites:

PRQ.1 a memory corruption to divert control flow;

PRQ.4 a writable buffer;

PRQ.5 the addresses of useful functions or ROP gadgets.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Note that prerequisite **PRQ.4** is easier to satisfy than **PRQ.2**. W⊕X cannot prevent any of the prerequisites for code reuse and, therefore, does not help against code-reuse attacks.

2.4.3 Memory safety & Control-flow integrity

One way to thwart control-flow hijacking is to prevent memory corruptions, i.e., trying to achieve memory safety. Memory safety defenses typically instrument code to maintain metadata about the bounds of memory objects. When pointers to memory objects are used for reading or writing, the metadata is checked to ensure that the read or write operation does not go out of bounds.

Another way to prevent control-flow hijacking is to tolerate the memory corruption, but prevent the adversary from diverting the control flow. By diverting the control flow, an attacker creates control-flow transitions not present in the original program. A mitigation technique called Control-Flow Integrity (CFI) tries to restrict the control flow to benign control-flow transitions [Aba+05]. The major challenge with CFI is determining a precise Control-Flow Graph (CFG) for the original program because the security of CFI directly depends on the precision of the CFG. Since the first proposed practical implementation of CFI [Aba+09], numerous refinements on the one hand and attacks against CFI on the other hand have been proposed [Dav+14; NT14; Car+15; Eva+15a; Con+15; Dav+15a; Gök+14; Mas+15; Ge+16].

Preventing memory corruptions or restricting the control flow targets prerequisite **PRQ.1**. Both memory safety and CFI open a whole new dimension of attacks and defenses and an in-depth discussion of the topic is out of scope for this work. We refer the interested reader to the relevant literature [Bur+17].

2.4.4 Code layout randomization

An important observation from the previous section is that an attacker needs to predict the addresses of functions or gadgets in the attacked process (prerequisite **PRQ.5**). Without randomization, predicting an address poses no problem because the memory sections of a process (see Section 2.1.2) start at predictable locations. The locations remain identical even across re-instantiations of the process. A defense called Address space layout randomization (ASLR) randomizes the base address of the process memory sections[3]. Hence, code addresses are shifted by a random offset and the attacker must resort to guessing the addresses. A graphical representation of ASLR is shown in Figure 2.8. On 32-bit systems, however, the entropy of randomization is limited by the size of the address space and ASLR can be bypassed with brute-forcing [Sha+04]. On 64-bit systems the need for position-independent code may hinder the practicability of ASLR [Pay12].

[3]The `text` section is only randomized for position-independent executables

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

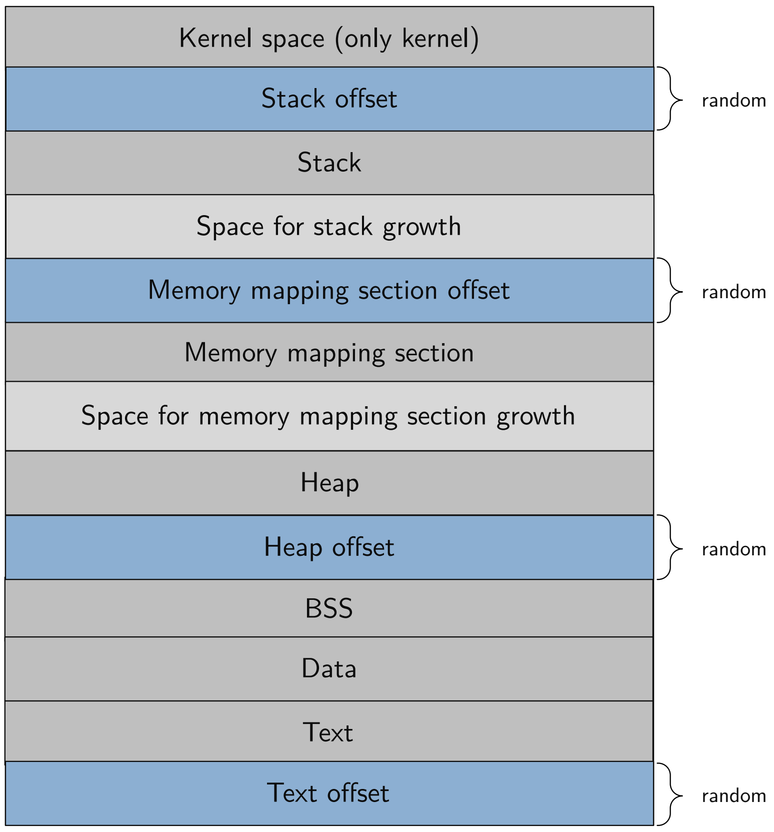


Figure 2.8: Memory layout with ASLR.

Even when ASLR is active for all parts of the process, adversaries can use *memory disclosure attacks* to leak information about the randomized code. As ASLR randomizes on a per-module basis, a single leaked pointer from a module suffices to figure out the random offset for that module. An attacker can then construct a ROP attack with code from the module offline and relocate the attack using the learned offset. In order to limit what attackers can learn from a leaked pointer, increasingly sophisticated fine-grained randomization techniques have been proposed [**Pappas2012**; Kil+06; War+12; Dav+13; Hom+13].

2.4.5 Just-In-Time Code Reuse

All fine-grained code randomization techniques share a common assumption: the attacker is not able to read the randomized code. An attack called *Just-In-Time Code Reuse* invalidated this assumption [Sno+13]. By using a memory disclosure vulnerability multiple times to disclose the randomized text section, attackers can reverse the effects of any code randomization scheme. The adversary discloses and disassembles one code page at a time. Any direct code reference found in the disassembled code (e.g., direct jumps or calls) leads to more pages that can be disclosed. The authors demonstrated the attack

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

with ROP as the code reuse mechanism and called it Just-In-Time ROP (JIT-ROP). Furthermore, an attack called Blind ROP (BROP) shows that not even an out-of-bounds pointer is necessary to disclose memory. In BROP an attacker uses brute-force to guess an initial gadget that invokes the `write` system call to disclose the text section.

Prerequisites A successful code-reuse attack against a process with code randomization assumes the following prerequisites:

PRQ.1 a memory corruption to divert control flow;

PRQ.4 a writable buffer;

PRQ.6 a memory disclosure vulnerability to disclose the text section.

2.4.6 Execute-only memory

A key observation from the previous section is that in the presence of code randomization the text section becomes a secret. If the attacker manages to disclose this secret, code randomization as a defense against code reuse becomes ineffective.

Benign processes typically only read code from the text section during instruction fetch. An adversary who wants to leak code on the other hand, needs to perform a regular read. A defense called *Execute-only memory* makes use of this fact by making the text section execute-only. Execute-only memory removes prerequisite **PRQ.6** for a code-reuse attack. Unfortunately the CPU design of current CPUs does not distinguish between *read* and *execute* permissions in the page table. As a result execute-only memory must be implemented at least partly in software.

Different proposals exist on how to emulate execute-only memory. The major challenge with execute-only memory is to separate code from data because compilers produce programs where code and data can occur on the same page (e.g., jump tables). A defense called XnR uses a sliding window of readable pages to allow access to code and data at the same time [Bac+14]. Another defense, called Readactor, separates code and data using a modified compiler [Cra+15b]. Furthermore, Readactor uses a CPU feature called *extended page tables* to improve the performance of execute-only memory. Yet another approach is to separate code and data at runtime on-demand. When data in the text section is read it is moved to a new location and the location in the text section is overwritten with random data [TSS15; Wer+16]. Future reads at the same address are redirected to the new data location. If the attacker tries to divert control flow to a location in the text section that was previously read, the CPU executes the random data and the program crashes.

2.4.7 Indirect JIT-ROP

Given execute-only memory, adversaries have resorted to new ways of undermining code randomization. While execute-only memory can prevent attackers from leaking the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

code itself, it cannot protect information about the code in other parts of the memory. A process contains different kinds of code pointers in readable memory sections. For example, each function call stores the address of the instruction after the call onto the stack (return address). Another example are function pointer variables on the stack. C++ programs additionally have vtables. vtables are tables containing function pointers used to implement polymorphic method calls and are stored in the data section of a process (see Section 2.1.2). All these pointers point to the text section, but are considered data. Consequently they must be readable. The more fine-grained the code randomization is, the more observations an attacker needs in order to mount a JIT-ROP attack. However, too fine-grained randomization prevents memory page sharing and increases the runtime overhead [Lar+14]. As a result, weaker randomization schemes like function permutation are used. We call such a randomization *medium-grained randomization*. Medium-grained randomization enables another type of attack called *indirect JIT-ROP* [Cra+15b]. In indirect JIT-ROP the attacker first uses pointers found in readable parts of the memory to locate functions in the text section. Next, she can mount a regular JIT-ROP attack.

A successful indirect JIT-ROP attack against a process with execute-only memory and medium-grained randomization assumes the following prerequisites:

List of prerequisites

PRQ.1 a memory corruption to divert control flow;

PRQ.4 a writable buffer;

PRQ.7 a memory disclosure vulnerability to disclose code addresses through pointers on the stack, heap or in the data section.

Note that prerequisite **PRQ.7** is easier to satisfy than **PRQ.5** because execute-only memory cannot protect the stack, data or heap section.

2.4.8 Address-Oblivious Code Reuse

Another way to attack applications protected by execute-only memory is Address-Oblivious Code Reuse (AOCR) [Rud+17]. In an AOCR attack the adversary takes snapshots of the readable execution state (e.g., the stack) of an application at attacker-controlled times. In order to precisely time the snapshots, the authors introduce a technique called Malicous Thread Blocking (MTB). Next, the adversary compares the snapshots to an offline run of the application to infer which of the pointers found in the snapshot point to which functions. Knowing the pointers enables the attacker to mount a whole function reuse attack. Multiple function invocations can be chained with a technique called Malicious Loop Reuse (MLR). Note that AOCR circumvents any kind of code randomization because it reuses whole functions instead of ROP gadgets.

A successful whole function reuse attack against a process with execute-only memory and fine-grained randomization assumes the following prerequisites:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of prerequisites

PRQ.1 a memory corruption to divert control flow;

PRQ.4 a writable buffer;

PRQ.8 pointers at known locations on the stack, heap or in the data section.

2.4.9 Summary of observations

In this section we shortly summarize important observations about the attack prerequisites described in the previous sections. First, note that all presented attacks share the common prerequisite

PRQ.1 a memory corruption to divert control flow.

We note that hijacking the control flow through a memory corruption is only possible because control-flow data and regular data are stored in the same memory. Preventing control-flow hijacking would thwart all of the presented attacks. However, techniques to prevent control-flow hijacking are often incomplete [Car+15] or cause a significant performance impact [Sze+13]. Techniques like code randomization on the other hand tolerate the control-flow diversion, but limit what an attacker can achieve with it.

Another observation is that all attacks require a writable buffer, either in the form of

PRQ.2 a writable and executable buffer

or the reduced version, which is easier to satisfy:

PRQ.4 a writable buffer.

Removing the prerequisite of a writable buffer is difficult because most programs need buffers to store inputs or temporary data.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 3

State of the Art

In this chapter we give an overview of existing defense techniques against control-flow hijacking attacks and how they protect against information disclosure.

3.1 Code-Pointer Integrity

The basic idea behind Code-Pointer Integrity (CPI) is to separate code pointers from other data (e.g. buffers) [KSP14]. In a first step, CPI uses a static analysis to detect all sensitive pointers. Sensitive pointers are code pointers as well as pointers manipulating code pointers. Next, CPI moves sensitive pointers to a secret area in memory, called safe region, and stores metadata describing the objects on which the pointers operate. To keep the metadata accurate, CPI instruments all instructions operating on sensitive pointers. The instrumentation makes sure that all instructions use only pointers stored in the safe region. The instrumented instructions check and update the metadata when pointers are dereferenced or manipulated. Pointers in the memory sections visible to an attacker represent offsets into the safe pointer store instead of real addresses. In addition to CPI, the authors propose a mechanism called *safe stack* to improve the performance of stack accesses. The safe stack is also located in the safe region and contains all code pointers and variables that can be guaranteed to be accessed in a safe way. Potentially unsafe variables are moved to a separate stack. CPI prevents **PRQ.1** as well as any kind of memory disclosure (**PRQ.5**, **PRQ.6**, **PRQ.7**, **PRQ.8**).

On x86-32 CPI uses hardware segment protection to secure the safe region. Accesses to the safe region happen solely through a dedicated segment register. On x86-64 CPI relies on ASLR to hide the safe region. The authors argue that the safe region is leak proof because no pointers visible to the attacker point into the safe region. A proof of concept attack on Nginx using a timing side-channel shows, however, that the safe region can still be leaked [Eva+15b]. Furthermore, an attack called *stack spraying* can be used to attack the safe stack [Gök+16]. As soon as an adversary knows the location of the safe region, she can use a memory corruption vulnerability to overwrite the protected pointers.

3.2 Readactor

Readactor is a security framework consisting of code randomization, execute-only memory and Code-Pointer Hiding (CPH) [Cra+15b]. CPH aims to decouple code pointers from the randomized code layout to make sure that attackers cannot use code pointers for indirect memory disclosure attacks. With CPH enabled, Readactor redirects code pointers through a table of execute-only trampolines. An execute-only trampoline consists of a branch instruction (`jmp` or `call`) encoding the pointer target address. As instructions are stored in execute-only memory, an attacker can no longer see the pointer target address, but only the address of the trampoline. To prevent an adversary from inferring the code layout through the trampolines, Readactor randomly permutes the trampoline table. To discourage guessing the code layout with brute force, Readactor inserts booby traps into the trampoline table. A booby trap is a piece of code lying dormant during normal program operation. If an attacker hits a booby trap, however, it actively responds to the ongoing attack. For example, a booby trap might analyze the current attack and lock down the attacked process [Cra+13].

Readactor can protect direct and indirect function calls as well as jumps to code addresses. In „It's a TRaP“ the authors further extend the concept of CPH to vtables [Cra+15a]. `Call` instructions need special treatment though because they automatically push the return address onto the stack. To protect return addresses, Readactor replaces call instructions with a jump to a special call trampoline. A call trampoline consists of a call instruction followed by a jump back to the original call site. When calling a function through a call trampoline, the address of the call trampoline is pushed onto the stack instead of the original code address. Figure 3.1 gives an overview of call trampolines. The authors report that call trampolines are responsible for the major CPH performance overhead. The performance overhead caused by call trampolines has two reasons:

1. Call trampolines are more common than trampolines for function pointers:

   A call trampoline is needed for each function call whereas trampolines for function pointers are only needed for address-taken functions.

2. Call trampolines lead to CPU pipeline stalls:

   When following a control flow transfer, the CPU fills the pipeline with instructions at the predicted branch target. For `ret` instructions, the CPU fills the pipeline with the instructions after the corresponding `call` instruction. However, with call trampolines the only instruction after the call is a `jmp` instruction, leading to the original call site. After executing the jump, the CPU must flush the pipeline and refill it with the instructions at the original call site. For a more detailed discussion of branch prediction and pipelines see Section 2.3.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

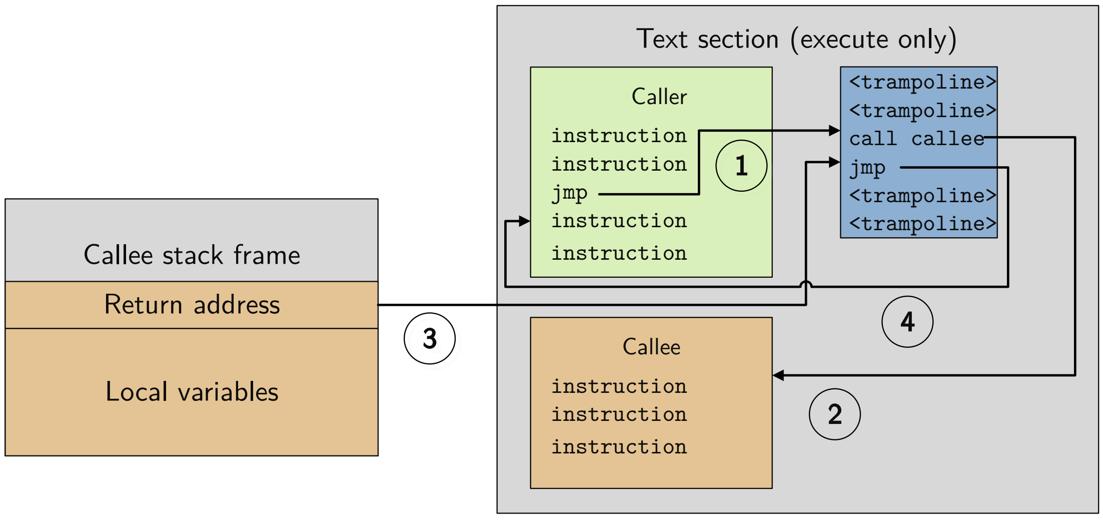


Figure 3.1: Hiding return addresses with call trampolines. 1. calls are replaced with jumps to a trampoline table; 2. a call in the trampoline table calls the callee; 3. the return address on the stack points to the trampoline table; 4. a jump in the trampoline table jumps to the real return address.

Readactor protects against direct and indirect memory disclosure attacks by removing prerequisites **PRQ.5**, **PRQ.6** and **PRQ.7**. However, as the redirected code pointers are still visible to the attacker, Readactor does not protect against **PRQ.8**. Therefore, AOCR attacks are still possible.

3.3 Isomeron

The idea behind Isomeron is to combine code layout randomization with execution-path randomization [Dav+15b]. Isomeron keeps two copies of the program, called *Isomers*, in memory. One of the Isomers is diversified with fine-grained code randomization. Isomeron ensures that no gadget is found at the same location in both Isomers. Each function f in the non-diversified Isomer has a semantically equivalent function f' in the diversified Isomer, but the functions f and f' have different code layouts. On each function call, a component called *execution diversifier* randomly decides whether to execute the called function in the diversified or non-diversified Isomer. In any case, the diversifier ensures that the return address on the stack always points to the non-diversified Isomer. However, due to code randomization, the diversifier must ensure that a called function always returns to the Isomer it was called from, independent from the return address on the stack. Therefore, the diversifier records its random decision during a function call. Upon return of a function, the diversifier retrieves the stored decision to continue execution in the originally calling Isomer. Ignoring the return address on the stack has an unwanted side-effect though. Whenever the return address on the stack does not match the calling function (i.e., the callee was called from the diversified Isomer), the CPU mispredicts

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

the return branch. In such a case the CPU fills the pipeline with instructions from the non-diversified Isomer and has to flush the pipeline when the diversifier transfers the control flow to the diversified Isomer. For a more detailed discussion of branch prediction and pipelines see Section 2.3.

Isomeron tolerates disclosure of the diversified and non-diversified Isomer, but still protects against ROP and JIT-ROP attacks. The idea is, that a gadget location is only valid in one of the Isomers. An attacker does not know in which Isomer the diversifier will continue execution upon return. Consequently, the attacker has to decide for one Isomer for each gadget. If the diversifier continues execution in the other Isomer, the attack would likely crash the program. Isomeron does not depend on ASLR or execute-only memory and removes part of prerequisite **PRQ.5**. Isomeron does not, however, prevent whole function reuse attacks because it randomizes the execution-flow on a function-granularity level. An attacker could still harvest code pointers by observing the stack or the heap. Therefore, Isomeron does not remove prerequisite **PRQ.7**.

3.4 CodeArmor

CodeArmor uses code randomization as a defense against control-flow hijacking and continuously changes the code's position to protect against information disclosure. To realize the position change, CodeArmor introduces an indirection between code pointers and their targets. In particular, CodeArmor prepares two regions in memory, the *virtual code space* and the *concrete code space*. The virtual code space initially contains the loaded program and code pointers point solely to the virtual code space. The virtual code space's position stays fixed. However, CodeArmor instruments benign (i.e. present in the original program) control-flow instructions to translate the virtual code addresses to concrete code addresses based on an offset V. The offset V determines the distance between virtual code space and concrete code space. If an attacker uses an unaligned code sequence to divert the control flow, the virtual code address is not translated and leads to the virtual code space. CodeArmor also ensures that the return addresses on the stack are virtual code addresses. Consequently, all addresses visible to the attacker (e.g., return addresses on the stack) are virtual addresses, but the process executes code in the concrete code space. The concrete code space contains the original program with a randomized code layout. To protect against information disclosure, CodeArmor continuously changes the position of the concrete code space and updates V. V is stored in a hidden memory location. To make sure, that an adversary cannot execute code in the virtual code space, CodeArmor fills the virtual code space with booby traps.

CodeArmor prevents direct and indirect memory disclosure by removing prerequisites **PRQ.5** and **PRQ.7**. While CodeArmor does not directly remove **PRQ.6**, an attacker would have to guess the location of the code to disclose because the concrete code space is constantly moved. The authors argue that CodeArmor also tolerates a disclosure of V because the concrete code space is moved with a high frequency and consequently also V changes frequently. However, CodeArmor protects only partly against whole

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

function reuse because it does not remove prerequisite **PRQ.8**. An attacker could use control-flow instructions present in the original program (i.e. instructions instrumented by CodeArmor), but with a different virtual addresses.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 4

Design

In this chapter we describe the conceptual building blocks of our solution against information disclosure, called Return Address Decoys (RAD). Our defense focuses on preventing indirect memory disclosure, in particular the disclosure of return addresses on the stack. First, we introduce our assumptions and establish a threat model. Next, we discuss the design decisions at the core of our defense. Finally, we show how the idea behind RAD also applies to C++ vtables.

4.1 Threat model and assumptions

Our defense builds upon other defenses to be effective. In particular, we assume the following existing defenses to be present:

- The target system provides protection against code injection. In modern processors and operating systems W⊕X is the de facto standard against code injection attacks;
- The target system provides fine-grained or medium-grained code layout randomization; The code layout randomization can be implemented in form of ASLR, compiler modifications or with load-time approaches like selfrando [Con+16];
- The target system provides execute-only memory. Pure software implementations as well as hardware-assisted implementations are available.

Furthermore, we assume the following threat model:

- The attacker cannot tamper with our customized compiler. Applications compiled with our custom compiler are fully equipped with RAD;

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- We cannot rule out the existence of timing, cache or fault side channels, but dealing with side channels is out of scope for this work.
- Due to code layout randomization, the attacker has no a priori knowledge of the target process's code layout;
- The attacker knows the system and software configuration of the target system. In particular, she has access to the source code of the target application and knows which defenses are in place;
- The application suffers from a memory corruption vulnerability that allows the attacker to modify any memory page writable by the target process;
- The application suffers from an information disclosure vulnerability that allows the attacker to read any memory page readable by the target process.

4.2 Overview of RAD

Before describing the design decisions behind RAD, we give an overview of how the defense works. RAD consists of a custom compiler which instruments the compiled program to insert fake addresses before and after the real return address in a stack frame. An example of RAD is shown in Figure 4.1. To discourage brute force attacks, the compiler uses booby trap addresses as fake addresses. Note that with RAD the correct return address remains in plain sight of the attacker, yet he does not know which of the addresses is the correct one. The secret of which addresses are fake is encoded in the instructions manipulating the stack and, therefore, protected by execute-only memory.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

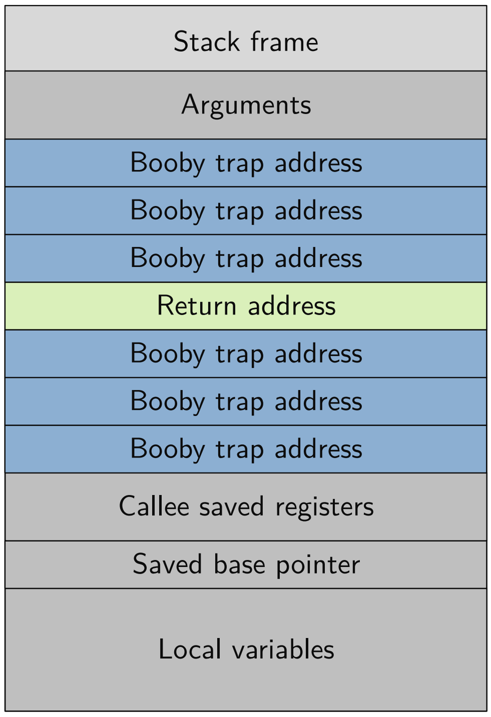


Figure 4.1: Example of a stack frame with RAD.

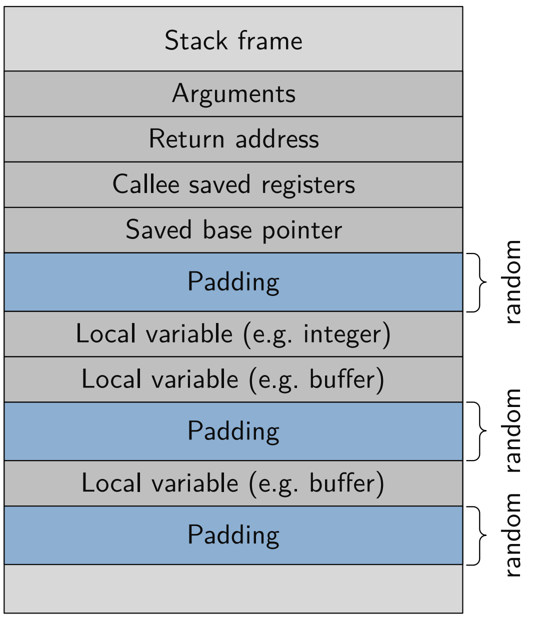
4.3 Design decisions

In this section we present the design decisions behind RAD. Each design decision was driven by analyzing the attack prerequisites presented in Section 2.4 and the shortcomings of solutions presented in Chapter 3.

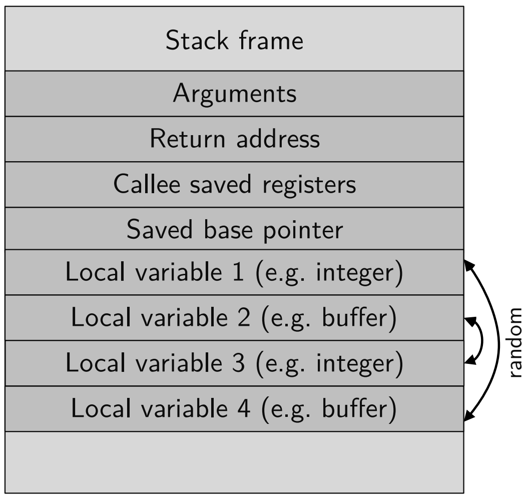
4.3.1 Software Diversity

The idea of software diversity is based on the observation that for a successful attack an adversary needs certain a priori knowledge about the target process. For example, for a ROP attack the attacker needs to know the exact memory addresses of gadgets. Software diversity aims to invalidate possible a priori knowledge by introducing artificial diversity into software. Artificial diversity means that implementation aspects are randomized while preserving the semantics of the application. For example, code layout randomization (see Section 2.4.4) is a form of software diversity. In particular, code layout randomization removes the a priori knowledge of memory addresses at which an attacker can find gadgets for a ROP attack. Due to its effectiveness against ROP attacks, we assume code layout randomization as a first line of defense. Without any additional defenses, however, code layout randomization cannot prevent JIT-ROP attacks (see Section 2.4.5). Even with defenses against JIT-ROP, indirect JIT-ROP and AOCR (see Section 2.4.7 and Section 2.4.8) remain a problem.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.



(a) Stack frame padding


Stack frame
Arguments
Return address
Callee saved registers
Saved base pointer
Local variable 1 (e.g. integer)
Local variable 2 (e.g. buffer)
Local variable 3 (e.g. integer)
Local variable 4 (e.g. buffer)
random

(b) Stack layout randomization

Figure 4.2: Examples of stack frame padding and stack layout randomization.

Indirect JIT-ROP and AOCR are built on the prerequisites

1. **PRQ.7** a memory disclosure vulnerability to disclose code addresses through pointers on the stack, heap or in the data section;

2. **PRQ.8** pointers at known locations on the stack, heap or in the data section.

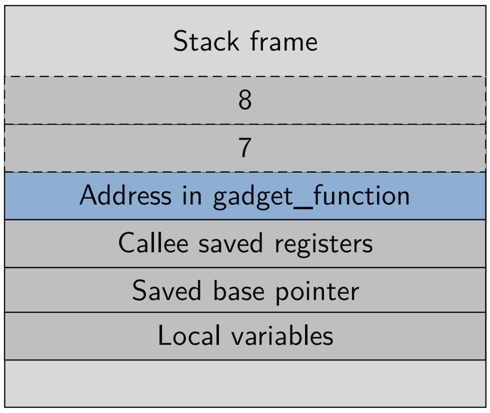
Note that prerequisite **PRQ.7** and **PRQ.8** contain an implicit assumption of a priori knowledge. For both prerequisites an attacker must know the location of code pointers on the stack or on the heap. Diversification techniques like stack frame padding or stack layout randomization introduce diversity into the layout of variables on the stack [Cra+15b]. Figure 4.2 shows an example of stack frame padding and stack layout randomization. However, with both techniques the base pointer and the return address stay at a fixed location, allowing the attacker to disclose them with a buffer over-read, for example. Even with a random padding between the buffer and the return address, an attacker could use pattern matching to locate the return address with the help of known neighboring values. For example, if the adversary knows the argument values of the function call, he could use them as an anchor to locate the return address. An example of this procedure is shown in Figure 4.3. The function `vuln_function` has fixed arguments, that are passed on the stack. An attacker can use these known arguments to locate the return address.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```c
void vuln_function(int a, int b, int c, int d, int e, int f, int g, int h) {
  // code, that contains a pointer leak
}

void gadget_function() {
  vulnerable_functiion(1, 2, 3, 4, 5, 6, 7, 8)
}
```

(a) Vulnerable code, which allows pattern matching.



(b) Stackframe of `vuln_function`.

Figure 4.3: Using pattern matching to locate the return address.

To address the a priori knowledge of a fixed return address location, we introduce random fake addresses on *both* sides of the return address. With fake addresses preceding and following the return address on the stack, pattern matching based on the known values would be unlikely to succeed. If an attacker tried to leak the return address, she would find a block of addresses each of which could be the real return address or a fake. An example of RAD is given in Figure 4.1.

We call the block of addresses surrounding the return address a RAD *configuration*. A RAD configuration is a tuple

$$\langle (f_0, .., f_i), (f_{i+1}, .., f_n) \rangle \, , \tag{4.1}$$

where $(f_0, .., f_i)$ denote the fake addresses before and $(f_{i+1}, .., f_n)$ the fake addresses after the return address. When a function with RAD is called, the resulting block of addresses on the stack looks as follows:

$$f_0, .., f_i, r, f_{i+1}, .., f_n \, . \tag{4.2}$$

The return address r depends on the function call site, while the fake addresses are generated by our customized compiler. Note that a single RAD configuration per function is sufficient to prevent an adversary from disclosing the return address with a single leaked stack frame. In the block of addresses in Equation (4.2) an adversary cannot differentiate the fake addresses from the return address. However, an attacker might observe the function stack frames when called from two different call sites. If two call sites used the same RAD configuration, the attacker would observe the following blocks of addresses on the stack:

$$o1 : f_0, .., f_i, r, \; f_{i+1}, .., f_n \tag{4.3}$$

$$o2 : f_0, .., f_i, r', f_{i+1}, .., f_n \tag{4.4}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

By comparing $o1$ and $o2$ an attacker could determine that only r and r' respectively have changed and, therefore, r and r' must be the real return addresses. To counter such deductions, the fake addresses differ not only for each function, but also for each call site. Let f_k^c be the fake address f_k of call site c and let C_F be the set of all call sites of function F. To prevent disclosure through stack frame observations, the following statement must hold:

$$\forall_{c_i \in C_F} \forall_{c_j \in C_F} c_i \neq c_j \implies \exists k : f_k^{c_i} \neq f_k^{c_j} \tag{4.5}$$

Given that different fake addresses are used for different call sites, our defense — with regard to return addresses — invalidates not only prerequisite **PRQ.7**, but also **PRQ.8**. If an attacker does not know the real return address, he cannot use it for whole function reuse attack.

4.3.2 Using a modified compiler

Different methods exist on how to diversify software in an automated way. Diversifying transformations are possible in any of the following stages:

1. pre-compile time;
2. compilation;
3. linking;
4. post-compile time;
5. loading;
6. runtime.

Pre-compile time transformations typically affect the source code of the program. For example, Cohen as well as Bhatkar, Sekar, and DuVarney explore source-to-source transformations [Coh93; BSD05]. However, as our technique changes an implementation detail of the calling convention, the required changes cannot be represented in source code. Post-compile time transformations are compatible with Commercial Off The Shelf (COTS) binaries (i.e., binaries where the source code is not available). Furthermore, load time transformations allow to re-randomize diversified parts on each load, and thus invalidate any knowledge an adversary might have gained from previous runs of the program. Runtime transformations like CodeArmor (see Section 3.4) even allow continuous re-randomization of the target process. However, the transformation from source code to a compiled program is a lossy transformation [Lar+14]. In particular, optimizations can obscure the original program structure. As a result, certain information about the program is not available anymore after the compilation process. For example,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

our technique modifies the stack layout and needs precise information about the layout of each stack frame (e.g., begin and end of each stack frame). Unfortunately, the stack layout information is not fully recoverable at runtime. For that reason, we opted for a compile-time transformation complemented by a link-time transformation. The compile-time transformation introduces the diversity described in the previous section, while the link-time transformation updates the call sites across all modules of the program.

4.3.3 Booby traps

In contrast to enforcement-based defenses like W⊕X or CFI, software diversity provides probabilistic security. Probabilistic security means that an attack can be prevented with a certain probability. A brute-force attack against software diversity might succeed by chance, similar to how an adversary might guess a password with brute-force. If a brute-force attack attempt crashes the target process, an attacker might retry modified versions of the attack, potentially using the information gained from previous runs. For example, an adversary could circumvent code layout randomization by guessing gadget addresses and retrying a ROP attack until he finds enough gadget addresses for a working attack. In the context of Rad, the secret an attacker is trying to find is the real return address among the fake addresses.

Just like a strong password statistically cannot be guessed within a lifetime, probabilistic security aims to decrease the chance of success for an attack far enough for an attack to become economically infeasible. One way to increase the cost for a successful brute-force attack is to increase the search space. If an attacker uses brute force to find a random secret, the size of the search space is given by the entropy underlying the random trial. For example, the search space for a randomly generated password can be increased by increasing the length of the password. For Rad, increasing the entropy in a single stack frame means increasing the number of fake addresses. The entropy across different call sites increases with the number of different Rad configurations per function. However, using more fake addresses has a negative impact on performance because the additional addresses must be written to the stack on each function call and also increase the size of the stack frame.

In situations where increasing the entropy is difficult, another approach is to penalize repeated attack attempts. If we used only random addresses as fake addresses, guessing the wrong address would cause the program to crash with a high probability because a subsequent ROP attack would be built based on incorrect assumptions (i.e., incorrect addresses). However, many network daemons, like the Apache web server, fork child processes and automatically restart the child processes if they crash, thereby giving rise to BROP attacks. As our solution does not include load-time re-randomization of the fake addresses, nothing would stop an attacker from repeatedly crashing the process while learning which of the addresses are fake and, therefore, steadily exploring the search space. To that end, we use booby trap addresses as fake addresses. If an attacker uses a booby trap address for a code-reuse attack, the attack not only fails, but the target process becomes aware of the ongoing attack and can react accordingly. In Rad, booby

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

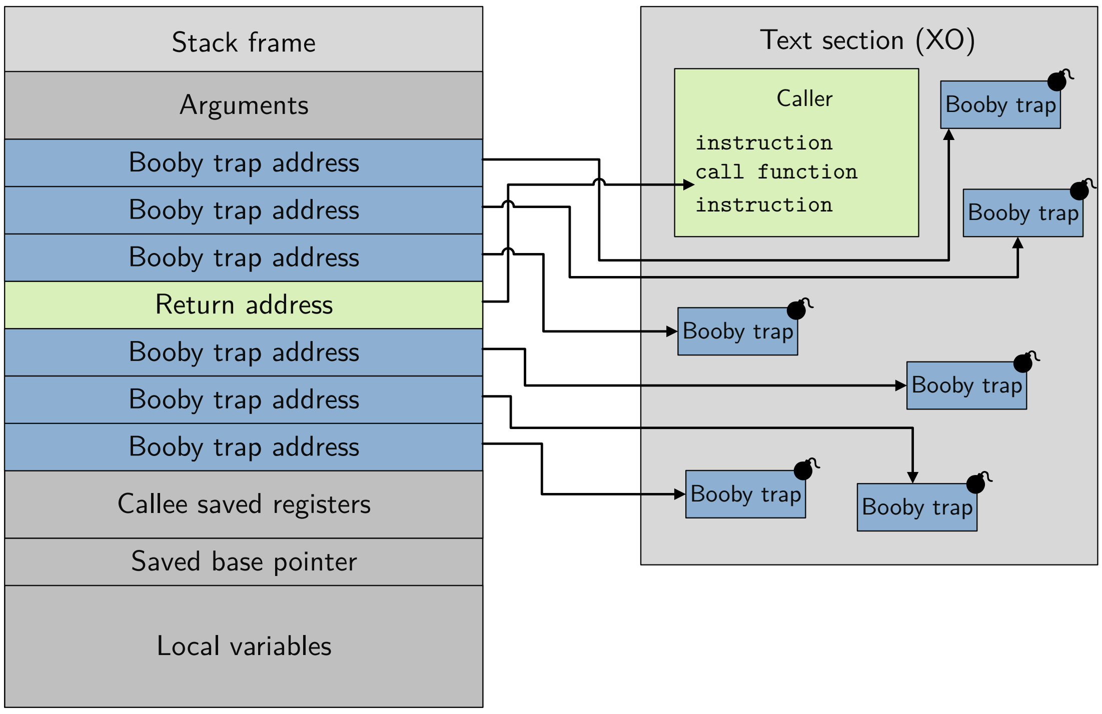


Figure 4.4: Booby trap addresses in Rad.

traps are addresses of booby trap functions. As a result, the fake addresses act as decoys luring the attacker into calling a booby trap function. Due to their role as bait, we call the fake addresses *return address decoys*. Figure 4.4 shows an example of booby traps in Rad.

Booby trap functions can respond in different ways to an attack. For example, a booby trap function could store the process's state for later analysis and revert the process to a known working state. An in-depth discussion of possible countermeasures is out of scope for this work and we refer the interested reader to the according literature [Cra+13].

4.3.4 Execute-only memory

Preventing attacks by taking away a priori knowledge from an attacker typically encodes the knowledge in information not known to the attacker. For example, code layout randomization removes the a priori knowledge of exact code addresses by randomizing the code layout. The secret consists of the random decisions driving the randomization process and is contained in the randomized instruction layout. Another example is CPI (see Section 3.1), which uses a secret memory region. Similarly, the diversifier in Isomeron (see Section 3.3) keeps a secret record of its decisions. In Rad, the secret is the location of the real return address in the block of fake addresses.

Different approaches exist on how to protect the introduced secrets. One approach is

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

to rely on information hiding. For example, CPI on x86-64 relies on ASLR to hide the secret memory region. However, attacks like stack spraying demonstrate that information hiding is not sufficient [Gök+16]. Another approach is to use enforcement-based defenses like execute-only memory. Readactor, for example, uses execute-only memory to protect the randomized code layout.

Rad is based on code layout randomization and code layout randomization needs protection itself. Execute-only memory has proven to be effective in protecting randomized code layouts. For that reason, we decided to use execute-only memory to protect both, the randomized code laoyut and Rad. Protecting a secret with execute-only memory means that the secret must be contained within the protected instructions. In Rad, the location of the real return address is encoded by instructions manipulating the stack pointer and by instructions writing the fake addresses to the stack.

4.3.5 Dealing with branch prediction

An important observation when analyzing Readactor (see Section 3.2) and Isomeron (see Section 3.3) is that both defenses cause a negative performance impact by disrupting the CPU's return address prediction. The branch predictor of modern CPUs uses the RSB to predict the target of function returns (see Section 2.3 for details). Note that the RSB operates independently from the stack in memory when recording the return addresses of function calls. Only `call` and `ret` instructions can modify the RSB. If the return addresses on the stack and in the RSB differ, the `ret` instruction will transfer control flow to the address on the stack, but the CPU will use the address in the RSB for its branch prediction. A similar issue occurs when the `call` instruction is immediately followed by another branch instruction, as is the case with return trampolines. With return trampolines the CPU fills the pipeline with the instructions after the trampoline instead of the instructions after the actual call site, leading to a pipeline stall. The CPU does not know that the return trampoline is only a temporary call site that transfers control to the actual call site.

To address the performance problems caused by pipeline stalls, Rad neither uses return trampolines nor modifies the return address on the stack. Instead, we leave the return address unchanged and protect it from disclosure with software diversity. Our modified compiler creates different function variants for different Rad configurations, so no trampolines are needed.

4.4 Protecting vtable pointers: VAD

The general idea of *multiplicity* behind Rad— to protect pointers by surrounding them with fake pointers — does not only apply to return addresses. Before designing and implementing Rad, we have explored the applicability of multiplicity to other sensitive code pointers. In particular, we designed a protection scheme for C++ vtables, called VTable Address Decoys (Vad). However, an implementation of both Rad and Vad was

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

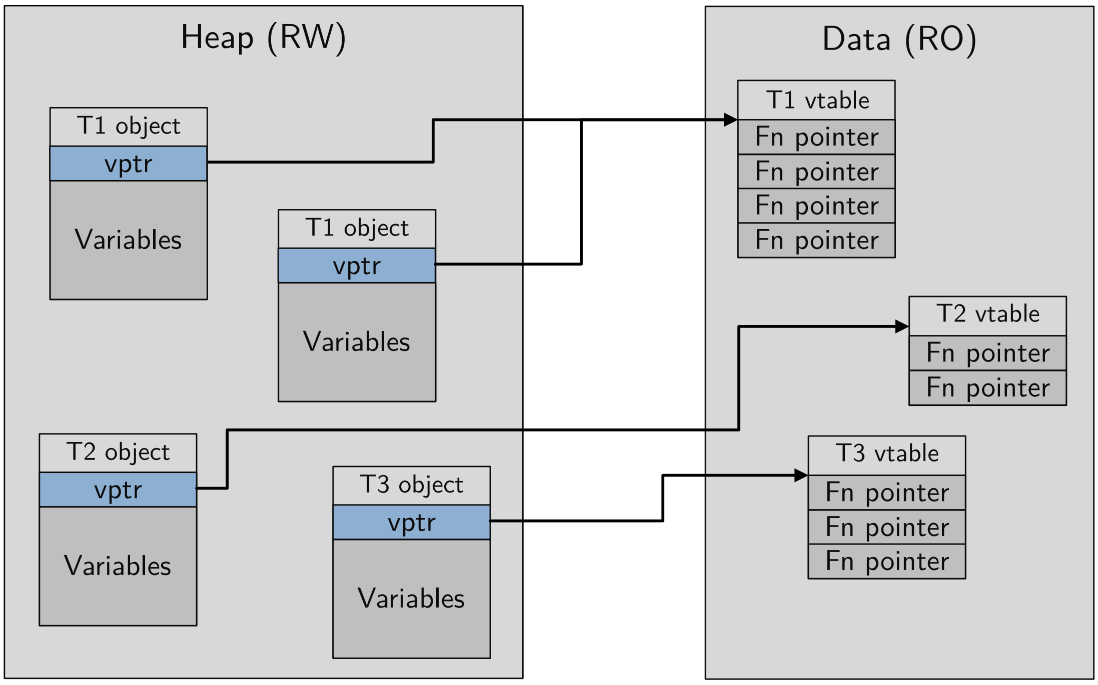


Figure 4.5: Example of objects and their vtables.

out of scope for this work and we implemented only a prototype of RAD. Furthermore, other defense techniques exist, which provide similar security guarantees to VAD [Tic+14; Cra+15a; Zha+16]. For the sake of completeness, in this section we give an overview of VAD, as it ultimately led to the design of RAD.

Vtables are an implementation technique for C++ polymorphic method calls. With a polymorphic method call on an object, the function to execute is determined based on the runtime type of the object. Instead of calling a function directly, the target function is looked up in the object's vtable. Vtables are tables containing function pointers to the functions belonging to a C++ class. On construction, each object receives a pointer to the vtable of its class. Vtables are stored in the read-only part of the `data` section, whereas objects are stored on the heap or the stack (see Section 2.1.2). An example of objects pointing to vtables is given in Figure 4.5. As vtables contain function pointers, they are a lucrative target for information leaks. Furthermore, an attack called Counterfeit object-oriented programming (COOP) uses vtables to mount a powerful code-reuse attack, which works despite defenses like code layout randomization and execute-only memory [Sch+15].

Our first approach was to surround benign vtables with randomly generated fake vtables. Similar to RAD, the fake vtables contain booby trap function addresses to look like real vtables. To prevent pattern matching, the benign vtables are permuted and also interspersed with booby traps. With VAD, an attacker leaking the `data` section can no longer differentiate the benign vtables from the fake vtables and as a result cannot use

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

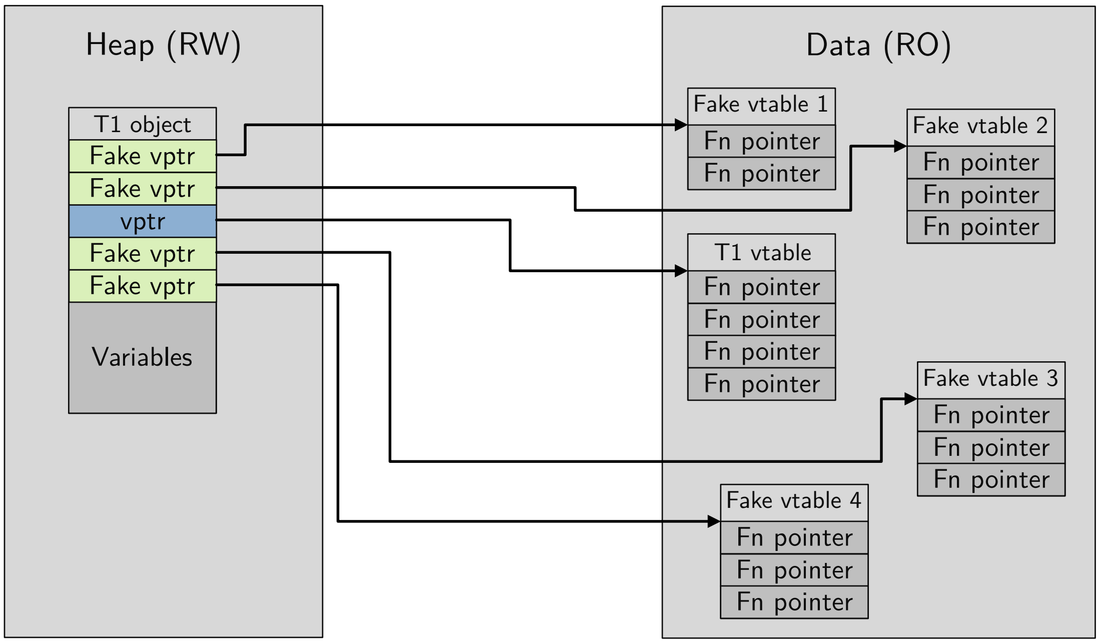


Figure 4.6: Example of an object protected by VAD.

the function pointers stored in the vtables. The advantage of this protection scheme is that it introduces *no* runtime overhead because the fake vtables lie dormant under normal program operation. However, the fake vtables can still be identified. Instead of leaking the `data` section, an attacker could resort to leaking C++ objects from the stack or the heap. As C++ objects contain pointers to the real vtables, they also reveal which of the vtables in the `data` section are fake. Identifying the benign vtables through pointers on the stack or the heap resembles an indirect memory disclosure.

In the presence of indirect memory disclosure, the challenge of protecting vtables becomes the challenge of protecting *pointers to vtables*. Consequently, we applied the idea of multiplicity also to vtable pointers. With VAD, each vtable pointer in an object is surrounded by fake vtable pointers, which point to one of the fake vtables. A graphical example of VAD is given in Figure 4.6. Analogous to RAD, we define a VAD configuration as a tuple $\langle (f_0, .., f_i), (f_{i+1}, .., f_n) \rangle$, where f_k are fake vtable pointers before and after the real vtable pointer. However, unlike RAD, the location of the real pointer is not determined by the function called. The callee at a virtual call site is generally not known at compile time[1]. Instead, the instructions at the call site use the vtable pointer stored in the object to lookup the callee in the corresponding object's vtable. As an example consider the virtual method call in Listing 4.1, Line 23. Listing 4.2 shows the corresponding assembly code. The assembly code first dereferences pointer `p` to obtain the vtable pointer and stores the result in register `rdx` (Listing 4.2, Line 2). Without

[1]Under certain circumstances the compiler can determine the callee of a virtual call site and replace the virtual call with a direct call. This optimization is called *devirtualization*.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
1  class Parent {
2  public:
3    virtual int age() {
4      return 35;
5    }
6    virtual int height() {
7      return 180;
8    }
9  };
10
11 class Child : public Parent {
12 public:
13   virtual int age() {
14     return 10;
15   }
16   virtual int height() {
17     return 120;
18   }
19 };
20
21 int getHeight(Parent *p) {
22   Parent *p = new Child();
23   return p->height();
24 }
```

Listing 4.1: Example of a C++ virtual method call.

```
1  ; rax contains pointer p
2  movq   0(%rax), %rdx   ; obtain the vtable pointer
3  callq  *8(%rdx)        ; call the second function in the
      vtable
```

Listing 4.2: Assembly on x86-64 of virtual method call in Listing 4.1, Line 23.

VAD, the vtable pointer is the first field in an object's memory representation, that is, the pointer to an object points to the vtable pointer. Next, the assembly code dereferences the vtable pointer and adds an offset of 8 bytes (i.e., size of one function pointer) because the function `height` is the second entry in the vtable of `Parent` or the vtable of `Child`, respectively.

With VAD, the assembly code needs to add an additional offset when obtaining the vtable pointer (Listing 4.2, Line 2) to select the real pointer among the fake pointers. An example of the code in Listing 4.2, but with VAD, is given in Listing 4.3. The code in

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
; rax contains pointer p
movq   24(%rax), %rdx  ; obtain the 4th vtable pointer
callq  *8(%rdx)       ; call the second function in the
    vtable
```

Listing 4.3: Assembly with VAD on x86-64 of virtual method call in Listing 4.1, Line 23.

Listing 4.3 assumes a VAD configuration with 3 pointers before the real vtable pointer and, therefore, adds an offset of 24 bytes (i.e., the size of three pointers) to obtain the 4th vtable pointer. Note that although the static type of `p` is `Parent*`, `p` can point to a `Child` object as well as a `Parent` object. As a result, objects of class `Child` and objects of class `Parent` must share the same VAD configuration. The class hierarchy in C++ forms a forest, and each class is part of a tree. The VAD configuration of a class is determined by its most generic predecessor, or by itself, if the class does not have a predecessor. In other words, all objects of classes in the same tree must share a VAD configuration because a pointer of type `T*` could point to an object of any subclass of `T`.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 5

Implementation

In this chapter we describe the implementation of RAD, our proposed defense against the disclosure of return addresses. As described in Section 4.3.2, we implemented our defense as part of a modified compiler. We chose to extend the LLVM compiler toolchain because LLVM is widely used, modular and open source.

5.1 Architecture of LLVM

LLVM (the acronym Low Level Virtual Machine has been officially removed) is a framework for implementing compilers and analysis tools. The LLVM website states:

> The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" itself is not an acronym; it is the full name of the project.

LLVM consists of several sub-projects, including the C and C++ frontend `clang` and the LLVM Core libraries. `Clang` is responsible for parsing C and C++ code and creates an intermediate representation from it. While some compilers use multiple intermediate representations, LLVM uses only one, called LLVM Intermediate Representation (IR). The IR is architecture agnostic and allows LLVM to express machine independent optimizations as IR transformations, which are typically realized as *passes*.

A pass is a C++ class performing a specific IR transformation. The LLVM Core libraries include numerous passes for different purposes. For example, the LICM pass performs the loop-invariant code motion optimization. While most of the included passes perform optimizations or analysis tasks, passes can transform the IR arbitrarily. LLVM separates a program into structures of different size, with modules being the top level structure.

A module consists of functions and each function consists of basic blocks. Passes can operate on any of these structures, as well as logical structures like loops.

After all IR transformations, LLVM converts the final IR into a Directed Acyclic Graph (DAG), called *`SelectionDAG`*. The `SelectionDAG` represents the control flow and data flow dependencies between the IR instructions. An example of a `SelectionDAG` is shown in Figure 5.1. Initially the nodes of the `SelectionDAG` are machine independent. LLVM uses a *backend* to progressively transform the nodes to concrete machine instructions, a process called *lowering*. A backend abstracts a specific machine architecture. For example, backends define the supported registers and machine instructions of the corresponding machine architecture. At the time of writing, the LLVM Core libraries include backend implementations for more than 40 machine architectures. Which backend is used during compilation, depends on the target machine. For example, when compiling a program for the x86-64 machine architecture, LLVM uses the x86 backend[1]. The result of the lowering process is a machine dependent `SelectionDAG`. Finally, the `SelectionDAG` is linearized to yield the machine code of the program.

Note that LLVM also implements an alternative way to lowering with the `SelectionDAG`, called *`FastISel`*. `FastISel` reduces the compilation time of a program at the cost of code quality (i.e., the resulting code is less optimized). `FastISel` only supports the lowering of a subset of the IR and falls back on the `SelectionDAG` otherwise. A third way of lowering, called *`GlobalISel`*, is supposed to replace both `SelectionDAG` and `FastISel`. However, at the time of writing the x86 backend had not been ported yet to GlobalSel.

Our modifications to LLVM consist of a module pass and modifications to the `Selec-tionDAG` lowering for the x86 backend. Therefore, the modifications are limited to the LLVM Core libraries and are compatible with any LLVM frontend. In the following sections we describe the module pass and the backend modifications in more detail.

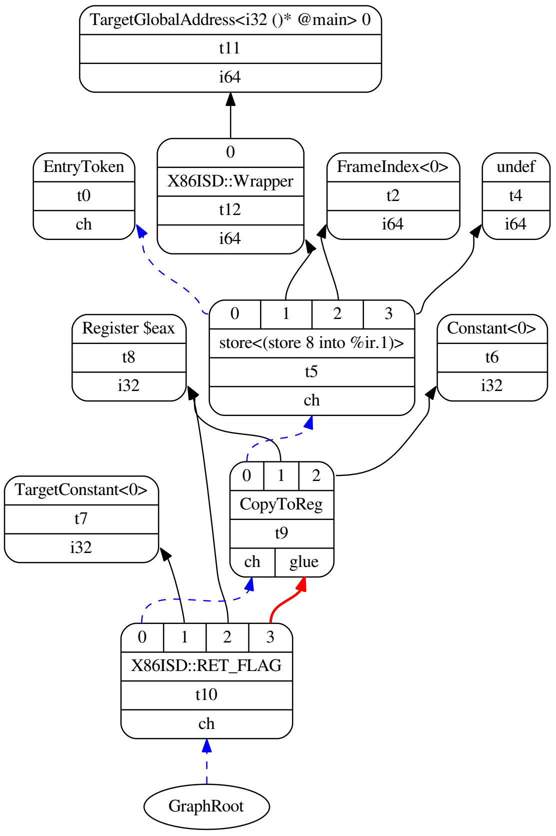
[1]The x86 backend in LLVM handles both machine architectures, x86 and x86-64.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

```
int main() {
  void *address = &main;
}
```

(a) Simple C program, which takes the address of a function.



(b) `SelectionDAG` of the program on the left.

Figure 5.1: A simple C program and it's `SelectionDAG` representation.

5.2 Diversity Pass

We decided to implement the machine independent part of our defense as a LLVM pass, called *`DiversityPass`*. Encapsulating functionality in a pass has several advantages. First, a pass can be distributed independently from the LLVM binaries and does not have to be compiled into LLVM, but can be loaded dynamically if required. Second, separating machine independent code from machine dependent code allows reusing the pass with different backends. Third, while a lot of resources exist on how to write a LLVM pass, little information is available on how to modify other parts of LLVM.

The `DiversityPass` runs on each module and performs the following tasks:

1. insert a booby trap function;

Algorithm 5.1: Diversify IR module

```
Input: An IR module M
Output: A diversified IR module M'
1  for $F \in M$ do
2      $variants \leftarrow \emptyset$;
3      for $C_F \in F$ do
4          if $|variants| = MaxVariants$ then
5              $V \leftarrow randomly\ choose\ from\ variants$;
6          else
7              $V \leftarrow generateVariant(F)$;
8              $variants \leftarrow variants \cup \{V\}$;
9          end
10         $replace\ C_F\ with\ a\ call\ to\ V$
11     end
12 end
```

2. randomly generate function variants representing different Rad configurations for each function;

3. replace the original calls to a function with calls to randomly chosen variants.

The process of replacing call sites is shown in Algorithm 5.1.

The `DiversityPass` allows to customize the parameters influencing the diversification. In particular, the following parameters can be supplied as command-line arguments:

- upper and lower bounds for the number of fake addresses before and after the return address;
- the maximum number of function variants to generate per function;
- the maximum number of stub functions to insert.

Changing the diversification parameters allows to customize the program for different security and performance requirements.

5.2.1 Metadata attributes

The `DiversityPass` generates Rad configurations for each function in an IR module. To represent different Rad configurations, the `DiversityPass` clones the original function and adds metadata attributes describing a specific Rad configuration to each clone. Metadata attributes are a way to add information to IR units (e.g., to instructions and types). The attributes can be used by other passes or can influence the lowering phase.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

For example, LLVM's Type Based Alias Analysis (TBAA) uses metadata attributes to attach type information to IR instructions. In the case of RAD, the metadata is used to change the lowering in the x86 backend. The metadata attributes in RAD consist of two groups. One group describes the addresses before the return address and one group the addresses after the return address. The groups are represented by different prefixes in the attribute names, e.g., `diversity-rad-before`. Each group contains an attribute that holds the number of fake addresses and one attribute for each fake address. The attributes describing the fake addresses contain the names of the functions whose addresses are used as fake addresses. For example, the attribute `diversity-rad-before-fn0` with the value `BT-0` means that the address of function `BT-0` is used as the first fake address before the return address. The `DiversityPass` keeps the original function in the module to retain backwards compatibility.

5.2.2 Stub functions

As discussed in Chapter 4, the fake addresses should differ between call sites, point to booby traps, and should also provide enough entropy to hide the return address. However, typically there is only one booby trap function implementation. As a result, using the booby trap function address as fake address would miss the goal of different fake addresses for different call sites. Furthermore, using only a single fake address would not provide enough entropy. To increase the amount of available fake addresses, the `DiversityPass` inserts stub functions which call the booby trap function. A stub function consists of a single call instruction to call the booby trap function. As RAD builds on code randomization, the addresses of the stub functions are also randomized. The stub functions are similar to the trampolines discussed in Section 3.2. However, with RAD the stub functions are not called during regular program operation, but are only called if an attacker uses their addresses in an attack. An example of a stub function is given in Listing 5.1.

```
000000000040086a <BT_11>:
  push   %rax
  callq  4006e0 <BoobyTrap>
```

Listing 5.1: Assembly code of a booby-trap function stub.

For testing purposes, the `DiversityPass` inserts a simple booby trap function in each module. The booby trap prints a warning text and exits the program. Inserting a booby trap function automatically allows the compilation of test programs (e.g., benchmarks) without modifying existing build chains. The inserted assembly code is shown in Listing 5.2. Another way to provide the booby trap function would be, for example, to mark a function in the program source code with an attribute.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
00000000004006e0 <BoobyTrap>:
  push    %rax
  mov     $0x4009f4,%edi
  xor     %eax,%eax
  callq   400500 <printf@plt>
  callq   400510 <abort@plt>
  data16 data16 data16 data16 nopw %cs:0x0(%rax,%rax,1)
```

Listing 5.2: Assembly code of the inserted booby trap function.

5.2.3 Link Time Optimization

As the `DiversityPass` is a module pass it operates on one module at a time. During compilation, a module typically represents a translation unit of the source program. For example, LLVM collects the functions in a C file (in C a file is a translation unit) in one module. Per default, static analysis is also performed on a per module basis. The `DiversityPass` depends on static analysis results because it needs to find the call sites of a function to replace the original call with a call to a function variant. When static analysis operates on one module at a time, it does not find inter-module calls. As an example, consider the C program in Listing 5.3. LLVM constructs two modules from this program, one containing the function `main`, and one containing the functions `multiply` and `square`. Each module is compiled separately and the results are subsequently combined by the linker. When run on the `math.c` module, the static analysis returns only the call in `math.c` Line 6 as call site for `multiply`. As a result, RAD would not be able to protect intra-module calls like the call in `main.c` Line 2.

To deal with intra-module analysis, LLVM provides Link Time Optimization (LTO). LTO allows the compiler to interact with the linker and to apply selected analyses and transformations to the whole program instead of a single module at a time. With LTO enabled, LLVM produces object files containing IR bitcode instead of native object files. The linker resolves the symbols in the bitcode files, merges all bitcode files into a combined module and invokes LLVM's LTO passes, followed by code generation. As a result, LTO module passes see the whole program as one module and static analysis also returns intra-module calls.

LTO is realized as a plugin for the linker. We used the LTO plugin in combination with the `gold` linker. For a pass to run during link time, it needs to be registered in the `populateLTOPassManager` method of the `PassManagerBuilder` class. Furthermore, LTO needs to be enabled during compilation with the `-flto` flag.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

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

int square(int a) {
  return multiply(a, a);
}
```

(a) math.c

```c
int main() {
  return multiply(3, 5);
}
```

(b) main.c

Listing 5.3: C program consisting of two files.

5.3 The x86 Target

After the `DiversityPass` has diversified the IR by introducing function variants, LLVM lowers the IR to produce machine code using a backend. A function variant differs from the original function only by having additional metadata attributes. A backend supporting RAD needs to interpret the metadata attributes to produce different machine code for the variants. We extended the x86 backend to lower the function variants on the x86-64 machine architecture.

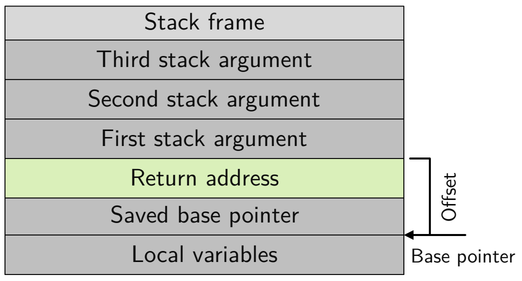
We decided to split the setup of the fake addresses on the stack between caller and callee. This decision is in line with the shared responsibility of stack frame setup during a function call (see Section 2.2 for details). When calling a function variant, the setup process looks as follows:

1. the caller performs all caller preparations and writes the function arguments to the stack (if any);
2. the caller writes the fake addresses before the return address to the stack;
3. the caller invokes the call instruction, which pushes the return address to the stack;
4. the callee reserves stack space for the fake addresses after the return address;
5. the callee saves the base pointer register on the stack;
6. the callee writes the fake addresses after the return address to the reserved stack space.

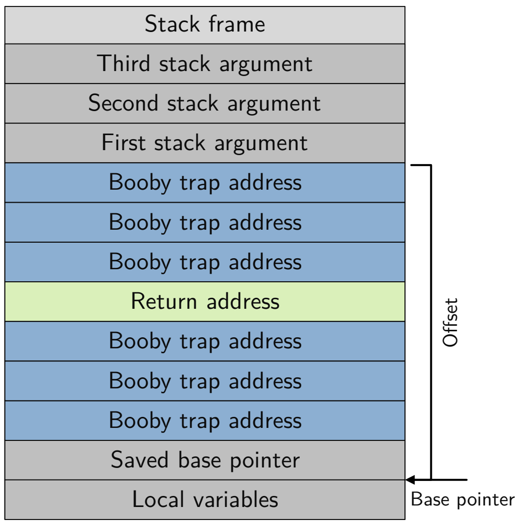
5.3.1 Setup in the caller

The setup in the caller is implemented in the `LowerCall` method of the `X86Target-Lowering` class. Certain methods in `X86TargetLowering` act as hooks, which the target independent part of LLVM calls during lowering to transform the `SelectionDAG` to a machine dependent representation. One of these hooks is `LowerCallTo`, which

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.



(a) Addressing arguments without fake addresses.



(b) Addressing arguments with fake addresses.

Figure 5.2: Examples of how stack arguments are addressed.

collects information about the call in a `CallLoweringInfo` object and then calls `LowerCall`.

We modified `LowerCall` to interpret the metadata attributes describing the fake addresses before the return address. `LowerCall` inserts nodes into the graph to resolve the function name from the attribute to an address and to write the address to the stack. To ease the implementation, `LowerCall` treats the fake addresses like arguments passed on the stack after the regular arguments. Treating the addresses as arguments has two important advantages. First, LLVM changes the calling function to reserve space in its stack frame. As stack arguments are written to the stack by the caller, the caller needs to reserve space in its own stack frame. Second, treating the fake addresses as arguments ensures that the callee addresses the regular arguments correctly. The callee typically adds an offset to its base pointer to address arguments passed on the stack. As fake addresses increase the distance between base pointer and the arguments on the stack, the callee needs to add an additional offset. Figure 5.2 shows examples of how arguments are addressed with and without fake addresses.

An example of a call to a function variant with fake addresses is given in Listing 5.4. To keep the example simple, the callee is a function without arguments or return values.

5.3.2 Setup in the callee

The setup in the callee is implemented in the `LowerFormalArguments` method of the `X86TargetLowering` class, as well as in the `emitPrologue` and `emitEpilogue` methods of the `X86FrameLowering` class. We modified `LowerFormalArguments`

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
movq   $0x4009fe,0x20(%rsp)
movq   $0x4009f8,0x18(%rsp)
movq   $0x4009f2,0x10(%rsp)
movq   $0x4009ec,0x8(%rsp)
movq   $0x4009e6,(%rsp)
callq  4009b0 <_Z4testv_Clone0_5_5>
```

Listing 5.4: Assembly code of a call which stores 5 fake addresses before the return address.

to write the fake addresses to the stack. `LowerFormalArguments` treats the fake addresses as a special case of arguments passed on the stack, similiar to `LowerCall`.

The methods `emitPrologue` and `emitEpilogue` are responsible for creating the function prologue or epilogue respectively. We modified `emitPrologue` to reserve stack space for the fake addresses in the callee. The space for the fake addresses is allocated before the prologue pushes the previous base pointer register to the stack. On the x86-64 machine architecture, stack space is allocated by decreasing the stack pointer. However, several instructions of the x86-64 machine architecture require the stack pointer to be 16 byte aligned. As the size of a 64 bit address is 8 bytes, allocating space for an uneven number of fake addresses leads to a stack pointer that is not aligned to a 16 byte boundary. To that end, `emitPrologue` inserts a padding to align the stack pointer if the number of fake addresses is uneven. An example of a function with a prologue that stores 5 fake addresses is shown in Listing 5.5. To keep the example simple, the function does not have any arguments or return values. Note how the `sub` instruction allocates 48 bytes instead of 40 bytes required by 5 fake addresses.

```
00000000004009b0 <_Z4testv_Clone0_5_5>:
  sub    $0x30,%rsp
  push   %rbp
  mov    %rsp,%rbp
  movq   $0x400a04,0x10(%rbp)
  movq   $0x400a0a,0x18(%rbp)
  movq   $0x400a10,0x20(%rbp)
  movq   $0x400a16,0x28(%rbp)
  movq   $0x400a1c,0x30(%rbp)
  pop    %rbp
  add    $0x30,%rsp
  retq
```

Listing 5.5: Assembly code of a function which stores 5 fake addresses after the return address.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

5.4 Limitations

Our implementation of Rad is a prototype implementation. While the prototype demonstrates the principles behind our defense, its implementation has certain limitations.

First, the function stubs consist of only a single instruction. If an attacker used one of the return addresses with an offset, he would not trigger the booby trap, but only crash the program. This problem can be addressed by designing more complex function stubs. One example would be to increase the size of function stubs with `nop` sleds. A `nop` sled is a series of `nop` instructions which increase the range of addresses leading to the booby trap function call. If an attacker jumped to any position in the `nop` sled, the CPU would first execute the `nop` instructions and would then call the booby trap function. However, when increasing the complexity of function stubs, it is important not to introduce gadgets an attacker could use for a ROP attack.

Second, we did not optimize our pass for compile time performance. Our pass invalidates all previous analysis results. As a result, LLVM needs to repeat the analyses after our pass has modified the module. The compile time performance could be increased by invalidating only those analysis results actually affected by our modifications. Moreover, LTO consumes a considerable amount of memory during compilation. LLVM provides a more efficient variant of LTO, called *`ThinLTO`*. However, passes need to be specifically designed for `ThinLTO` and the construction of a `ThinLTO` pass is considerably more complex.

Third, creating function variants by cloning the original function increases the code size of the resulting program. Note that the clones only differ in their function prologue and epilogue. The code size could be decreased by merging all basic blocks that do not contain the prologue or epilogue. LLVM already contains an optimization called *basic block merging*. However, basic block merging operates only within a function and the implementation of an inter-procedural basic block merging optimization was out of scope for this work.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 6

Evaluation

In this chapter we present the performance and security evaluation of Rad.

6.1 Performance

To assess the performance impact of our defense we compiled the integer benchmarks of the SPEC2006 benchmark suite with different diversity parameters. We chose SPEC2006 because it is commonly used in the scientific community to evaluate security defenses. Table 6.1 shows the diversity parameters for different benchmark configurations. When the minimum and maximum number of decoys is equal, the DiversityPass uses an exact amount of decoys, otherwise the numbers act as lower and upper bounds for a random trial. All configurations use a maximum of 100 stub functions. The benchmarks were executed in a LXC container with 24GB of RAM, running Debian 9 and the Linux kernel 4.4.134. The container was executed on a physical server with an Intel i7-2600 CPU running at 3.4 GHz. The benchmarks were compiled without optimizations enabled. All reported results are the average of 3 consecutive benchmark runs.

	Before		After	
name	min	max	min	max
baseline	0	0	0	0
rad_0_5	0	0	5	5
rad_5_0	5	5	0	0
rad_5_5	5	5	5	5
rad_5_5_random	2	5	2	5

Table 6.1: Diversity parameters for different benchmark configurations.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

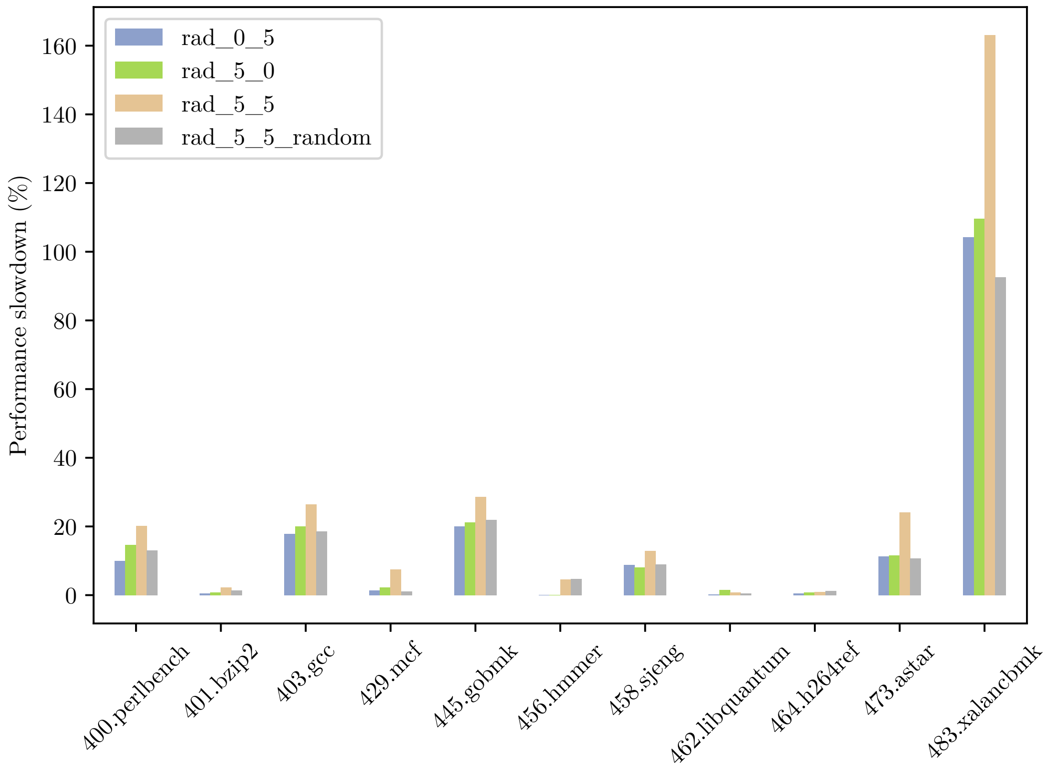


Figure 6.1: Performance slowdown in percent relative to the baseline build.

We executed the benchmarks with the LLVM utility `lnt`. `Lnt` executes programs based on simple test description files and compares the program's output with an expected output. The LLVM distribution already includes test description files for the SPEC2006 benchmark. To that end, `lnt` was not only useful to measure the runtime performance, but also to assert that our modifications do not break the program's behavior. Note that some of the tests contain multiple testcases. The shown results are aggregations of the testcase specific results for each benchmark.

6.1.1 Runtime overhead

We measured the runtime overhead by measuring the wall clock time a benchmark needs to finish. Figure 6.1 shows the slowdown compared to the baseline configuration. The absolute runtimes as well as figures comparing the absolute values for each benchmark can be found in Appendix A.1.

To better understand the cause of the performance impact we also analyzed each running benchmark with the Linux `perf` tool. `Perf` uses hardware performance counters to collect different metrics of a running program. In particular, we collected the number of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

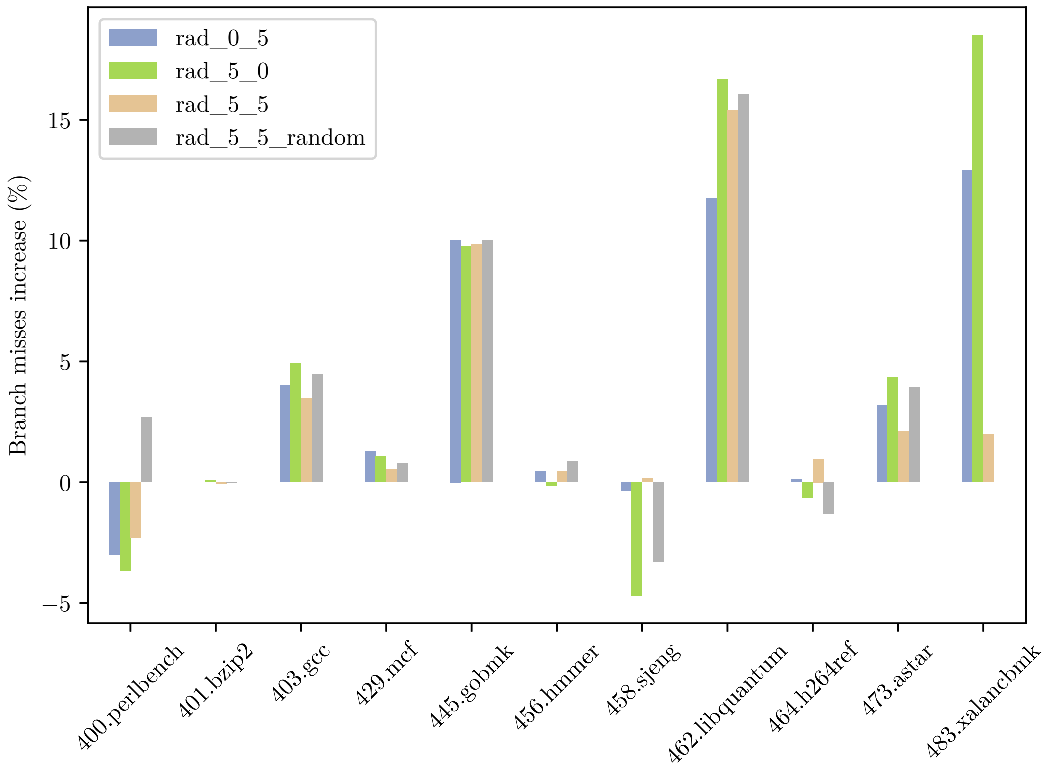


Figure 6.2: Increase in the number of branch misses in percent relative to the baseline build.

branch misses and the number of executed instructions. Figure 6.2 shows the increase in branch misses compared to the baseline configuration. Figure 6.3 shows the increase in instructions compared to the baseline configuration.

6.1.2 Memory usage

We assessed the memory usage of the different configurations by capturing the Proportional Set Size (PSS) of the running benchmarks every second and by calculating the average over the collected values for each benchmark. Capturing the memory usage of a process on a modern operating system is difficult for two reasons:

1. The operating system might page out parts of the memory used by a process;
2. Multiple processes might share libraries.

The PSS measure sums the private pages and the shared pages for each process and mitigates the second issue by dividing the number of shared pages by the number of

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

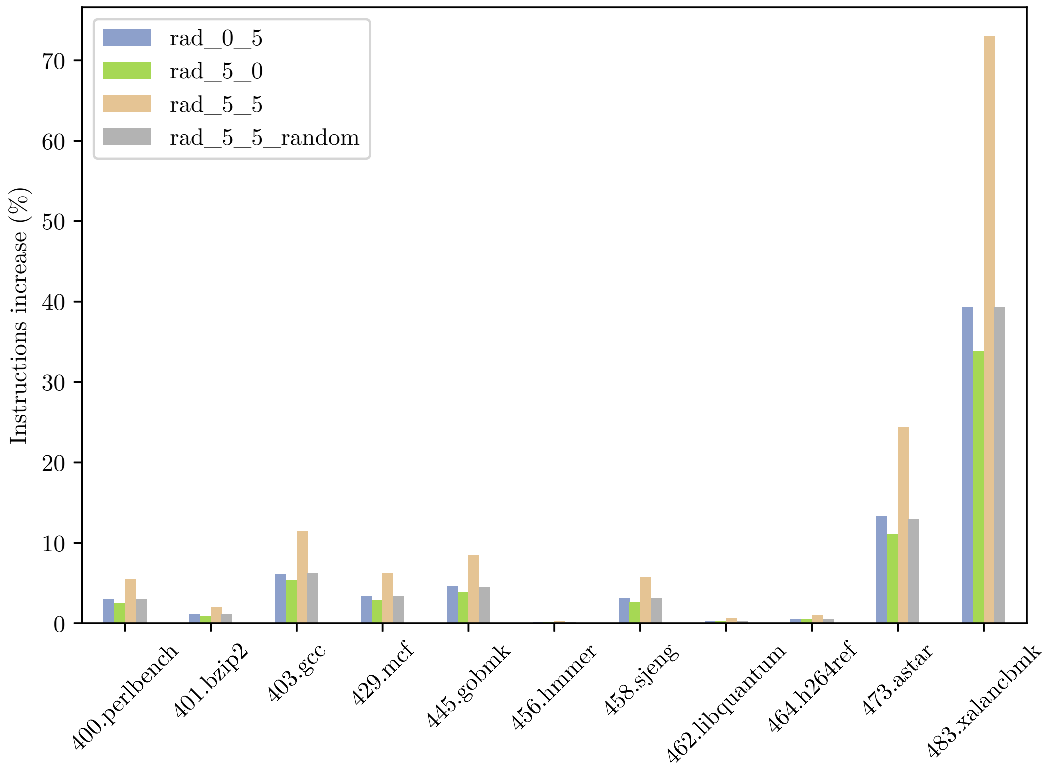


Figure 6.3: Increase in the number of instructions in percent relative to the baseline build.

processes sharing them. For example, if process A has 100 KiB of private memory, process B has 50 KiB of private memory and both share 200 KiB of memory, then process A has a PSS of 200 KiB ($100 + \frac{200}{2}$) and process B has a PSS of 150 KiB ($50 + \frac{200}{2}$).

We measured the increase of the PSS averages compared to the PSS averages of the baseline configuration. Figure 6.4 shows the PSS increase in percent compared to the baseline configuration. The absolute PSS values as well as figures comparing the absolute values for each benchmark can be found in Appendix A.2.

6.1.3 Code size

We measured the increase of the binary size by measuring the size of the compiled programs on disk and compared it to the size of the baseline configuration. Figure 6.5 shows the binary size increase in percent compared to the baseline configuration. The absolute code sizes as well as figures comparing the absolute values for each benchmark can be found in Appendix A.3.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

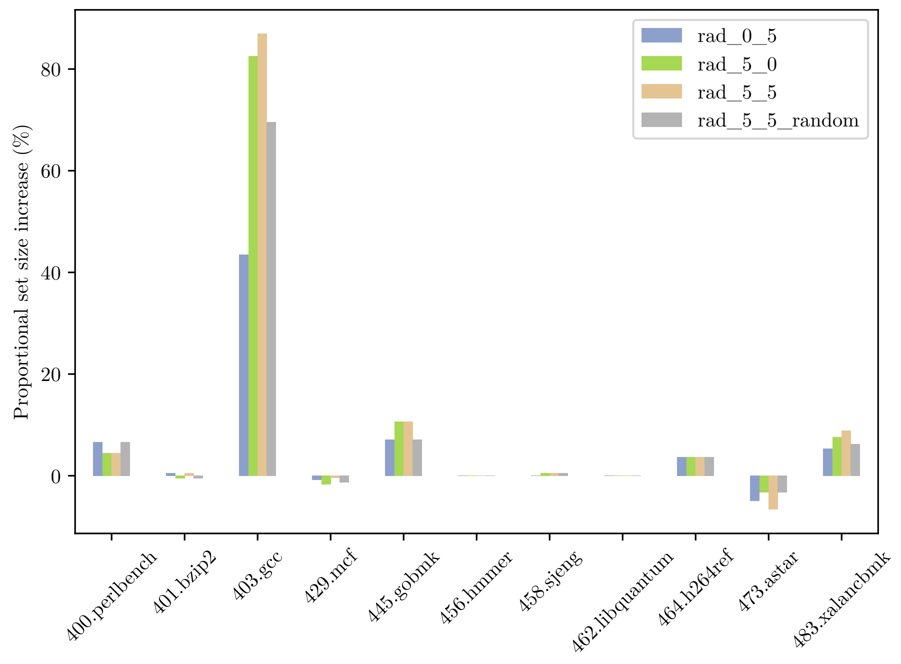


Figure 6.4: PSS increase in percent relative to the baseline build.

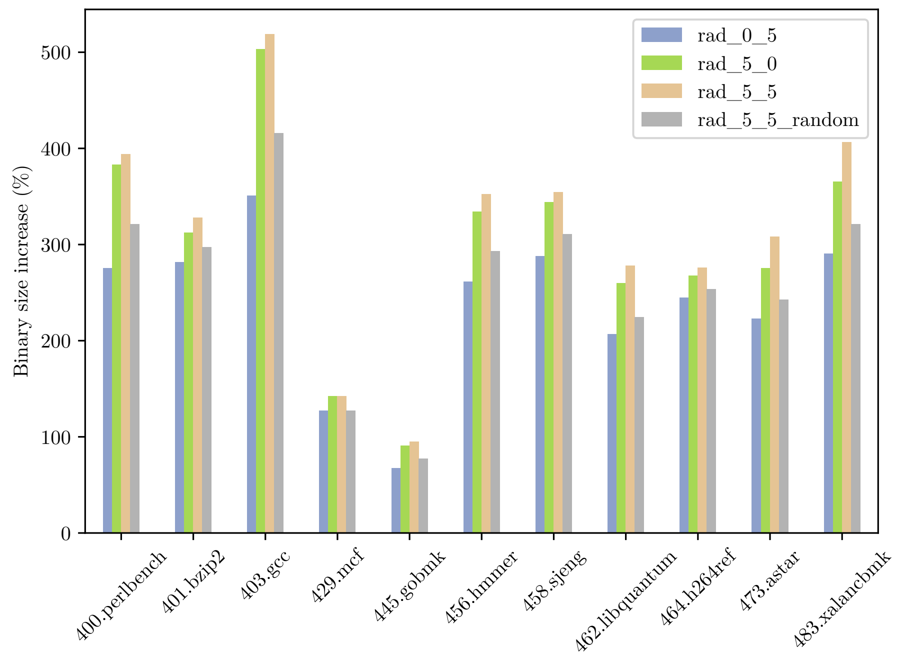


Figure 6.5: Binary size increase in percent relative to the baseline build.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

6.2 Security

The security of Rad is given by the probability with which an attacker can guess enough return addresses in leaked stack frames to mount a successful ROP attack. The probability to guess a return address is determined by the diversification parameters and by how many stack frames an attacker is able to leak.

Definition 1. *Let* $c = \langle (f_0, .., f_i), (f_{i+1}, .., f_j) \rangle$ *be a* Rad *configuration. Then* $|c| = i + j$ *is the number of decoys in the* Rad *configuration.*

We can analyze the probability by analyzing what an attacker can do with a single or with multiple leaked stack frames.

1. The attacker leaks a single stack frame.

   Let c be the Rad configuration of the leaked stack frame. We assume that the total number of stub functions is greater or equal to $|c|$ to avoid reusing addresses within a single stack frame. Therefore, the probability of picking the right address out of $|c| + 1$ addresses ($|c|$ decoys plus the return address) is given by

$$p_{single} = \frac{1}{|c| + 1} \, . \tag{6.1}$$

2. The attacker leaks stack frames from different call sites.

   Note that conceptually there is no difference whether the call sites belong to one or to different functions as the return address is always different. The only exception are recursive calls from the same call site, in which case the addresses in the leaked stack frames look the same and an attacker does not profit from multiple stack frames. Let n be the total number of call sites in the program. The call sites have the Rad configurations $c_1, .., c_n$. The total number of stub functions and, therefore, the total number of available decoy addresses is given by S. We assume that $|c_i| \leq S$ for all $i \in \{1, .., n\}$, which means that no decoys are reused within a single stack frame. We can now differentiate two sub-cases:

   a) $\sum_{i=1}^{n} |c_i| \leq S$
      If the total number of decoys in all Rad configurations is less than or equal to the total number of stub functions, no decoys are shared between the configurations. As a result, an attacker cannot match addresses between the stack frames, but must guess the return address in each stack frame individually.

   b) $\sum_{i=1}^{n} |c_i| > S$
      If the total number of decoys in all Rad configurations is greater than the total number of stub functions, Rad configurations from different call sites might share decoys. An attacker can use this information to narrow down

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

the correct return address within a single stack frame. If an attacker finds an address in two or more stack frames, he knows that the shared address cannot be a return address because no two call sites share the same return address. The probability of decoys overlapping is dependent on the size of the chosen RAD configurations. As the sizes of the configurations underlie a random trial, the sizes of different configurations can differ.

However, the probability of decoys overlapping is largest if all configurations use the maximum number of decoys. For simplicity, lets assume that all configurations uniformly use the maximum number of decoys m. Let $p_i(x)$ be the probability that with i leaked stack frames x decoy addresses can be matched. That is, we pick one of the i stack frames and try to find matching addresses with any of the other $i-1$ stack frames. We define $p_i(x)$ recursively as follows. The base case $i = 1$ is given by $p_1(0) = 1$ and $p_1(x) = 0$ for $x > 0$ because no addresses can be matched with a single stack frame. Let

$$p_i(x|y) = \frac{\binom{m-y}{x-y}\binom{S-(m-y)}{m-(x-y)}}{\binom{S}{m}} \tag{6.2}$$

be the conditional probability that with i leaked stack frames x addresses can be matched, given that with $i-1$ stack frames y addresses could be matched. We can apply the law of total probability to receive

$$p_i(x) = \sum_{y=0}^{x} p_i(x|y) * p_{i-1}(y)\,. \tag{6.3}$$

An attacker can use the matching technique described in Item 2 to reduce the number of possible addresses in Equation (6.1). If an attacker can match k addresses, he can reduce the size of the attacked configuration by k. We can now combine the probability to match decoys with the probability to guess the address within a single stack frame. Given that $|c_i| = m$ for all $i \in \{1, .., n\}$ (i.e., all configurations have the same size m) and the maximum number of decoys is S, the probability of finding a single return address with i leaked stack frames is given by

$$p_{address}(i) = \sum_{k=0}^{m} p_i(k) * \frac{1}{m-k+1}\,. \tag{6.4}$$

As a ROP attack typically requires multiple addresses, an attacker must repeat the process for each address to leak. The probability of finding n addresses with i leaked stack frames is given by

$$p_{attack}(n, i) = p_{address}(i)^n\,. \tag{6.5}$$

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 7

Discussion

RAD introduces instructions, that write decoy addresses to the stack and that are not present in an unprotected program. These instructions have are executed whenever a protected function is called. Therefore, the performance impact depends on the number of called functions. The benchmark `483.xalancbmk` is particularly affected because `483.xalancbmk` has a large number of function calls. The increase of executed instructions in function intensive benchmarks is visible in Figure 6.3.

We speculate that the slight increase of branch misses in some of the benchmarks is another source of slowdown. While differences of 5% or less can be attributed to measuring inaccuracies, Figure 6.2 shows that `445.gobmk`, `462.libquantum` and `483.xalancbmk` suffer from a higher increase in branch misses. While investigating the `483.xalancbmk` benchmark with `perf`, we found that most branch misses, in the baseline as well as all RAD configurations, are caused by intra-procedural unconditional jumps and direct calls. The CPU does not need to predict whether a branch is taken or not for unconditional jumps or direct calls. However, the CPU needs to decode the instructions to know that they are branch instructions. To allow fetching the correct instructions even before the branch instruction is decoded, the CPU uses the Program Counter (PC) as an index into a cache called Branch Target Buffer (BTB). The BTB contains a mapping between code addresses and branch targets. As RAD function variants introduce jumps at new code addresses, it also increases the pressure on the BTB. BTB misses due to a higher number of jumps could be the reason for the slowdown. However, BTB misses cannot explain why the number of branch misses is not higher for the `rad_5_5` configuration, which among all configurations introduces the most function variants.

The increase in memory usage of all but one benchmark is negligible. `473.astar` even uses less memory than the baseline, which indicates that the difference is due to measuring inaccuracies or caused by paging decisions of the operating system during the benchmark run. However, with RAD enabled, `403.gcc` used considerably more memory than the baseline. For example, the absolute difference between the baseline and `rad_5_5` is 20 MiB. We speculate that the increased memory usage is caused by the increased code size and a high locality of the code, which leads to most of the code residing in memory. The memory usage cannot be explained with the increased code size alone because otherwise `483.xalancbmk`, which has a similar code size increase, would suffer from a higher memory usage as well.

Rad affects the code size of a program for two reasons. First, Rad inserts stub functions to increase the number of booby trap addresses. Second, Rad creates clones of the original functions to create different Rad configurations. The increase in code size is directly proportional to the number of Rad configurations used as well as the number of functions present in the benchmark. We already discussed possible mitigations in Section 5.4.

We argue that the probabilistic security provided by Rad is enough to defend against information leaks even with multiple leaked stack frames and a small Rad configuration size. For example, the probability of leaking 10 return addresses with 30 stack frames, a maximum of 100 different decoy addresses and a Rad configuration size of 5 is still less than 1% (0.7133%). Furthermore, Rad is compatible with other defense techniques like CPH. For example, CPH could be used to protect function pointers whereas Rad is used to protect return addresses.

7.1 Limitations

Rad alters call sites to call function variants instead of the original function. The original function remains in the program to retain backwards compatibility with unprotected modules. However, altering call sites to call a different function means that the callee must be known at compile time. As a result, Rad cannot protect dynamic call sites. An exception to this problem are C++ virtual call sites. While the exact callee is not known at compile time, the compiler knows at least the most generic type of the calling object. Rad could use a similar approach to Vad (see Section 4.4) and assign a common Rad configuration to each type tree to mitigate the problem of unprotected C++ virtual calls. However, an implementation of this approach was out of scope for this work.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 8 Conclusion

In this thesis we explained the fundamental principles of how processes are represented in memory and how this representation enables code-reuse attacks. We discussed concrete implementations of ever more sophisticated attacks and identified issues with existing defenses.

We presented RAD, a technique to prevent return address leaks, which is based on the insights gained from analyzing the current attack and defense landscape. RAD uses software diversity to dramatically increase the efforts for an attacker to leak return addresses as preparation for a JIT-ROP attack. We also demonstrated how the idea behind RAD can be used to protect vtable pointers.

Based on the LLVM compiler suite, we implemented a configurable prototype of RAD. Different diversification parameters allow to fine-tune the compromise between security and performance. We evaluated the runtime performance, memory usage and code size increase of SPEC 2006 CPU benchmarks compiled with our prototype. We also discussed possible causes for the performance impact and gave a theoretical analysis of the security guarantees provided by RAD.

We found that RAD provides strong security guarantees, but can cause a considerable impact on runtime performance. The analysis of the performance impacts highlights possible areas for future improvements. We conclude that — although it is not a silver bullet — RAD further raises the bar for adversaries to mount a successful code-reuse attack.

8.1 Future work

Despite active research and a long list of proposed attacks and defenses, we believe that there are still many open questions to be answered. RAD is but one instantiation of a more general idea, whose applicability to other areas is not fully explored yet.

Rad protects return addresses against memory leaks. Furthermore, we have discussed the possibility to protect function pointers in C++ vtables. However, the stack as well as other readable parts of the memory can contain more types of code pointers an attacker can use for a JIT-ROP attack. For example, linkers on Linux use two tables, the Global Offset Table (GOT) and the Procedure Linkage Table (PLT), to realize position-independent code and dynamic linking of functions respectively. Both tables are readable at runtime and provide an attacker with an accumulation of code pointers. The idea behind Rad could be used to protect these tables from memory leaks.

We have implemented Rad as part of the LLVM compiler suite and as a protection scheme for statically compiled programs. However, Rad could also be useful during dynamic compilation in Virtual Machines (VMs) like the V8 javascript VM.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

APPENDIX A

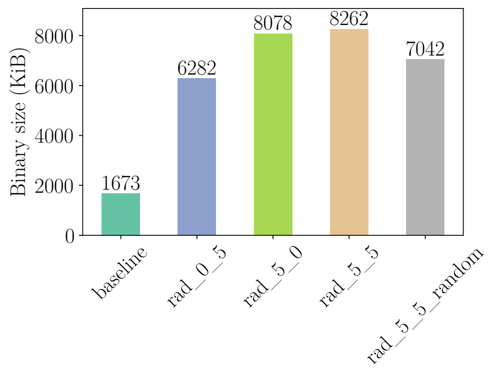
SPEC CPU 2006 Benchmarks

A.1 Runtime

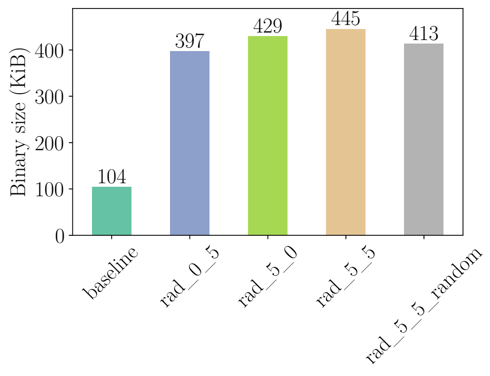
	baseline	rad_0_5	rad_5_0	rad_5_5	rad_5_5_random
400.perlbench	30.00	32.99	34.38	36.07	33.90
401.bzip2	81.17	81.56	81.81	83.01	82.25
403.gcc	1.40	1.65	1.68	1.77	1.66
429.mcf	18.94	19.21	19.36	20.36	19.16
445.gobmk	148.87	178.60	180.50	191.50	181.43
456.hmmer	124.44	124.42	124.33	130.17	130.28
458.sjeng	209.46	228.07	226.47	236.45	228.24
462.libquantum	3.84	3.85	3.90	3.87	3.86
464.h264ref	194.66	195.56	196.13	196.45	197.19
473.astar	149.66	166.64	166.92	185.75	165.70
483.xalancbmk	272.43	556.36	571.21	716.71	524.77

Table A.1: Runtime in seconds of the SPEC CPU 2006 benchmarks.

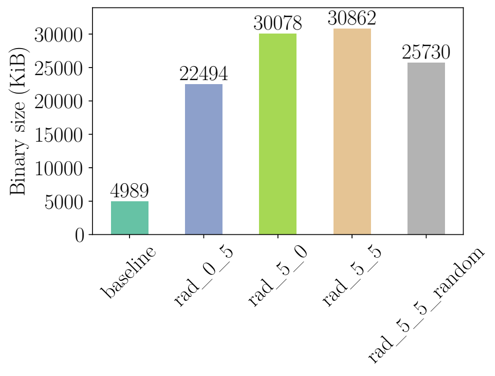
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.



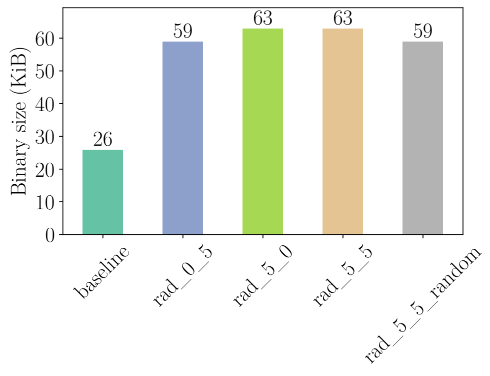
(a) 400.perlbench



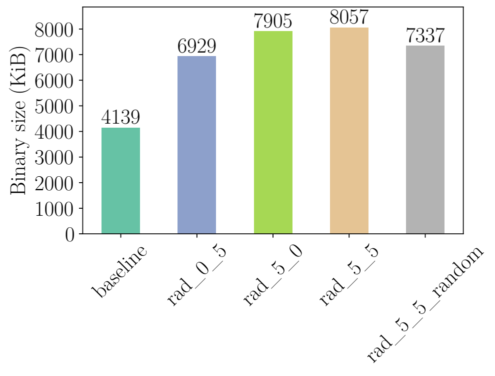
(b) 401.bzip2



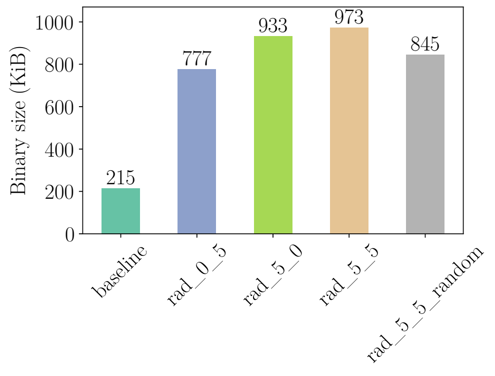
(c) 403.gcc



(d) 429.mcf



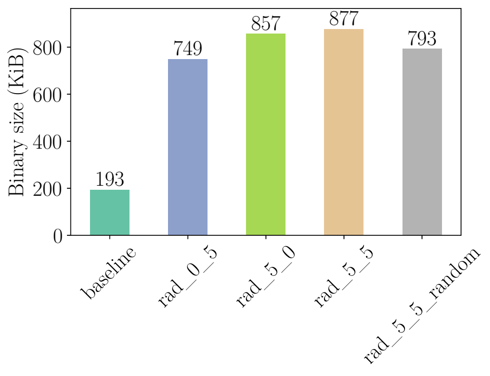
(e) 445.gobmk



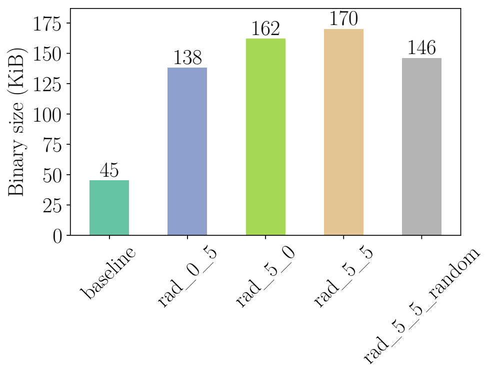
(f) 456.hmmer

Figure A.1: Runtime in seconds of the SPEC CPU2006 benchmarks per benchmark.

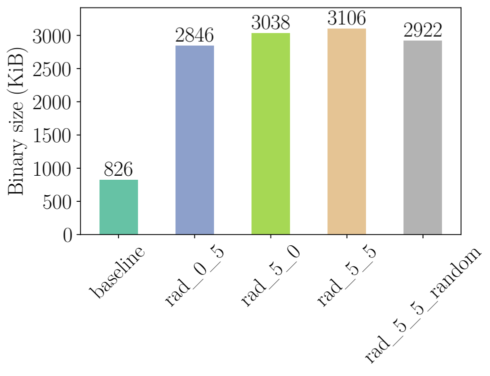
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.



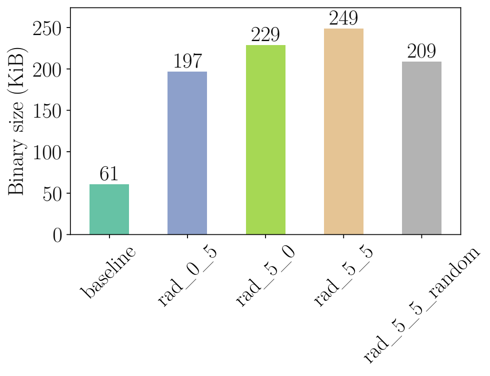
(g) 458.sjeng



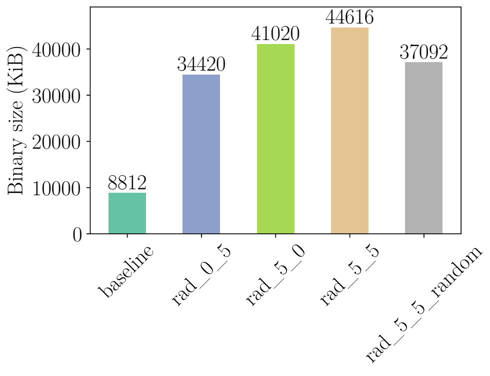
(h) 462.libquantum



(i) 464.h264ref



(j) 473.astar



(k) 483.xalancbmk

Figure A.1: Runtime in seconds of the SPEC CPU2006 benchmarks per benchmark.

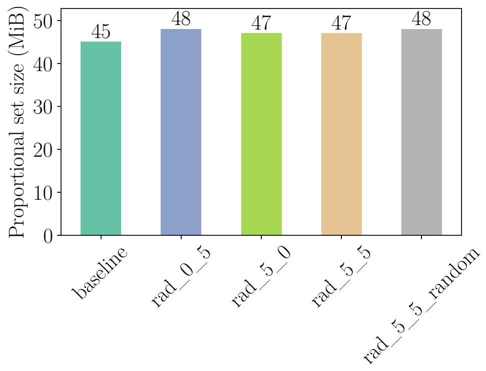
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

A.2 Memory usage

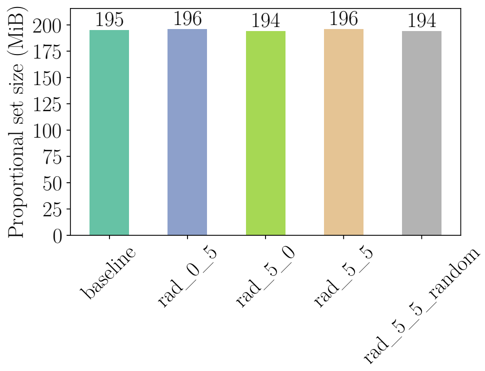
	baseline	rad_0_5	rad_5_0	rad_5_5	rad_5_5_random
400.perlbench	45.0	48.0	47.0	47.0	48.0
401.bzip2	195.0	196.0	194.0	196.0	194.0
403.gcc	23.0	33.0	42.0	43.0	39.0
429.mcf	231.0	229.0	227.0	230.0	228.0
445.gobmk	28.0	30.0	31.0	31.0	30.0
456.hmmer	3.0	3.0	3.0	3.0	3.0
458.sjeng	175.0	175.0	176.0	176.0	176.0
462.libquantum	1.0	1.0	1.0	1.0	1.0
464.h264ref	27.0	28.0	28.0	28.0	28.0
473.astar	60.0	57.0	58.0	56.0	58.0
483.xalancbmk	224.0	236.0	241.0	244.0	238.0

Table A.2: Proportional set size of the SPEC CPU 2006 benchmarks.

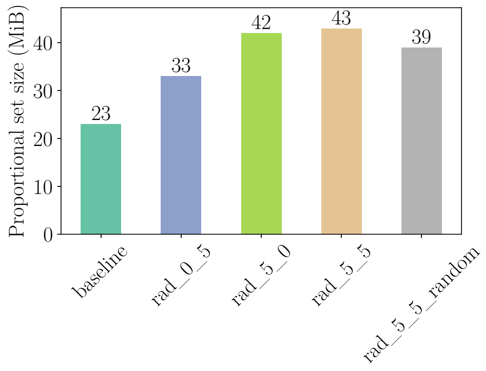
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



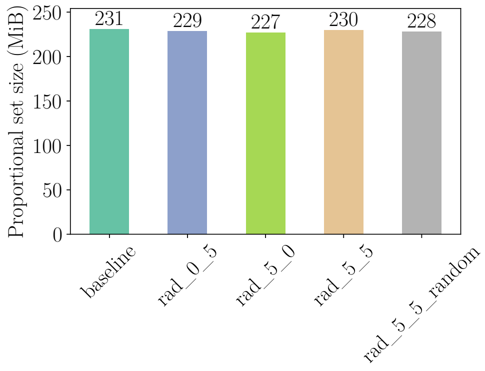
(a) 400.perlbench



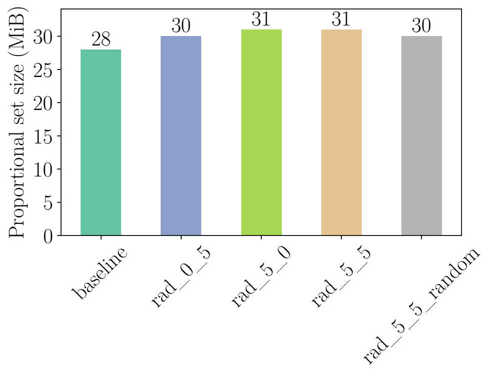
(b) 401.bzip2



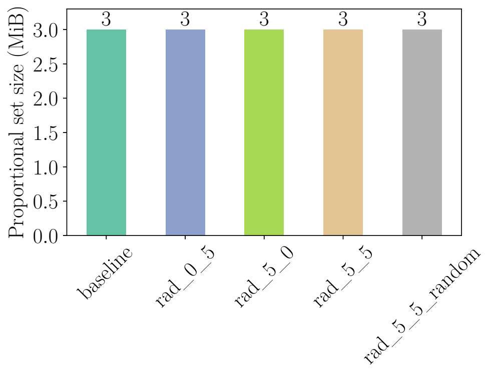
(c) 403.gcc



(d) 429.mcf



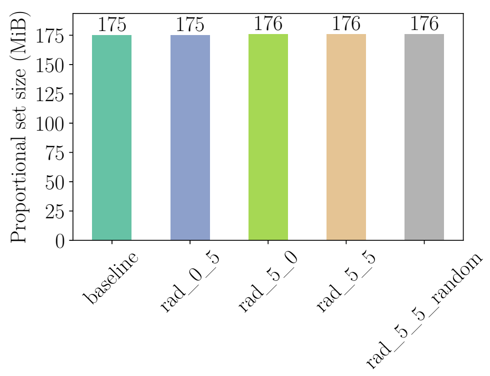
(e) 445.gobmk



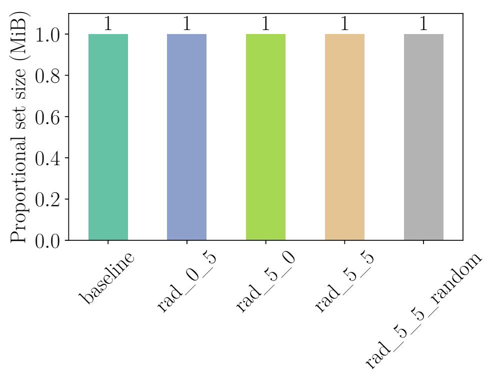
(f) 456.hmmer

Figure A.2: Proportional set size of the SPEC CPU2006 benchmarks per benchmark.

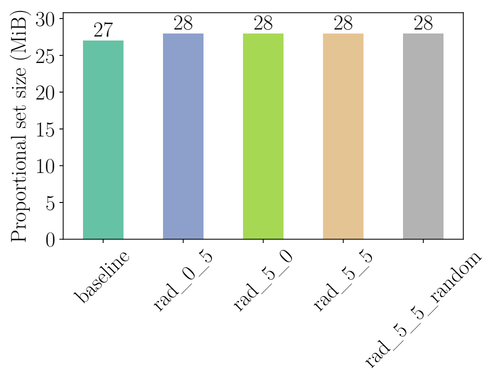
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.



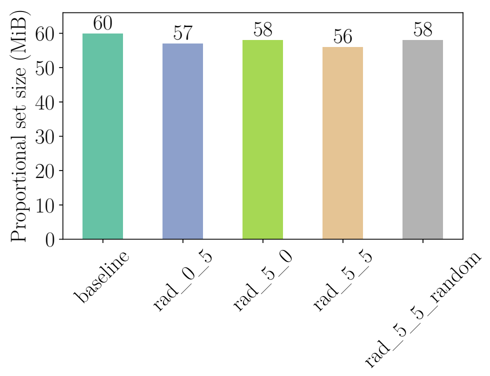
(g) 458.sjeng



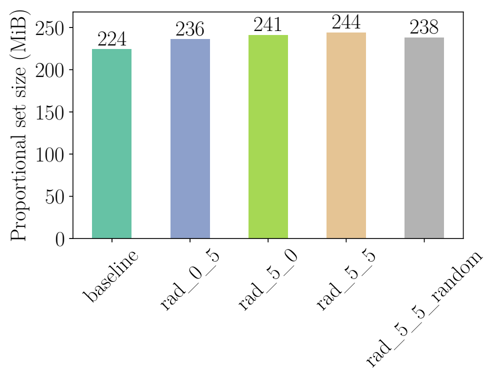
(h) 462.libquantum



(i) 464.h264ref



(j) 473.astar



(k) 483.xalancbmk

Figure A.2: Proportional set size of the SPEC CPU2006 benchmarks per benchmark.

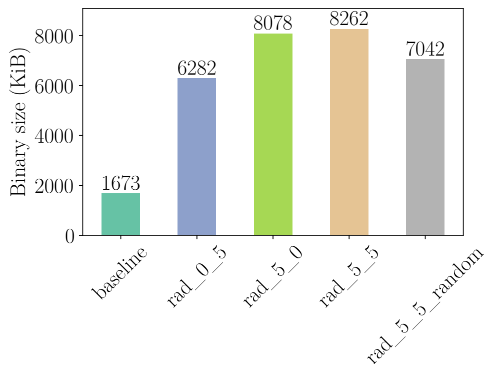
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

A.3 Code size

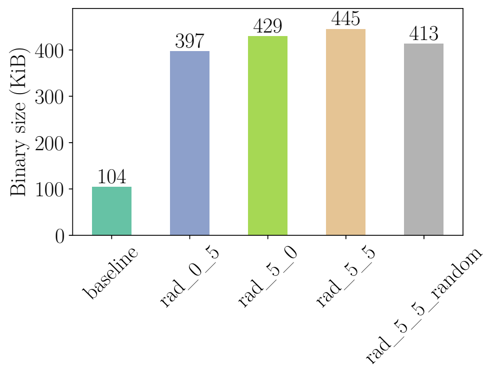
	baseline	rad_0_5	rad_5_0	rad_5_5	rad_5_5_random
400.perlbench	1673	6282	8078	8262	7042
401.bzip2	104	397	429	445	413
403.gcc	4989	22494	30078	30862	25730
429.mcf	26	59	63	63	59
445.gobmk	4139	6929	7905	8057	7337
456.hmmer	215	777	933	973	845
458.sjeng	193	749	857	877	793
462.libquantum	45	138	162	170	146
464.h264ref	826	2846	3038	3106	2922
473.astar	61	197	229	249	209
483.xalancbmk	8812	34420	41020	44616	37092

Table A.3: Binary sizes in KiB of the SPEC CPU 2006 benchmarks.

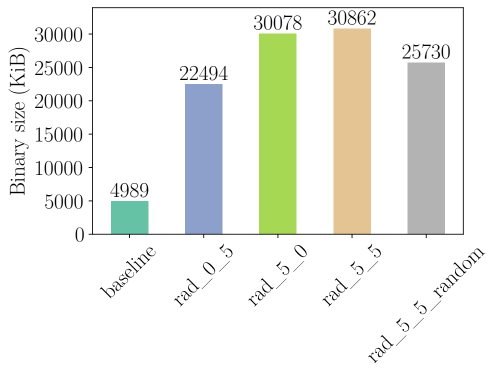
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



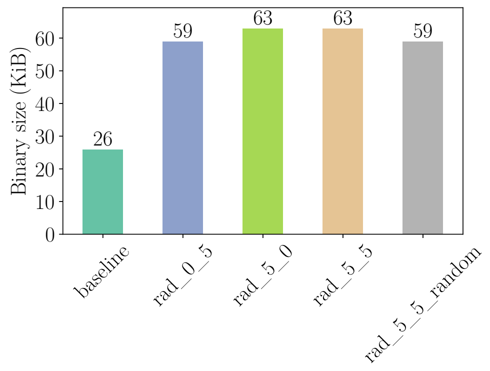
(a) 400.perlbench



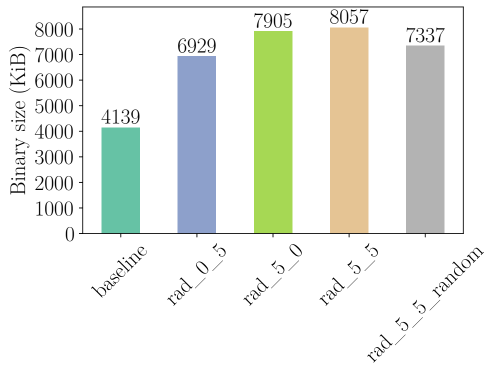
(b) 401.bzip2



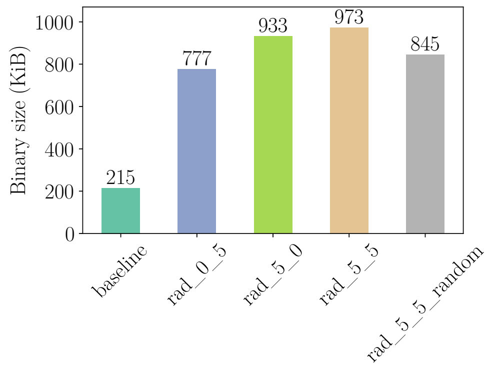
(c) 403.gcc



(d) 429.mcf



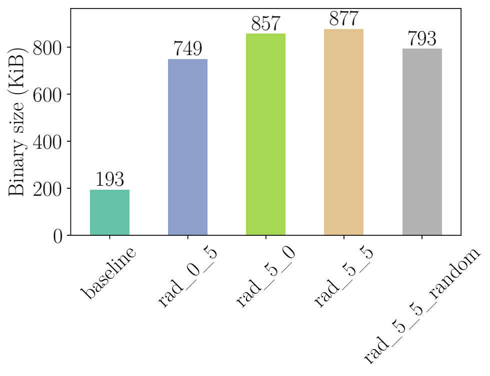
(e) 445.gobmk



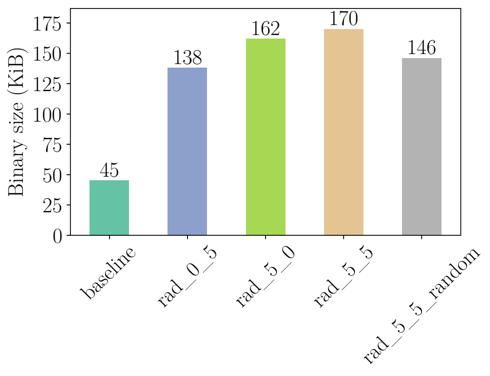
(f) 456.hmmer

Figure A.3: Binary size in KiB of the SPEC CPU2006 benchmarks per benchmark.

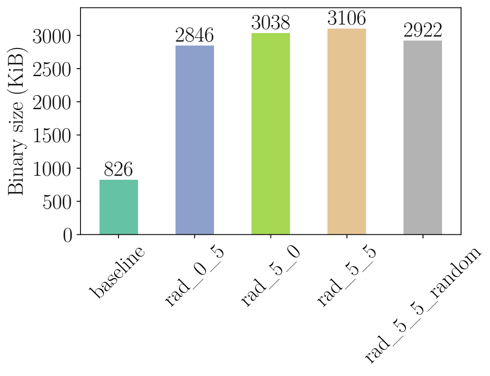
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.



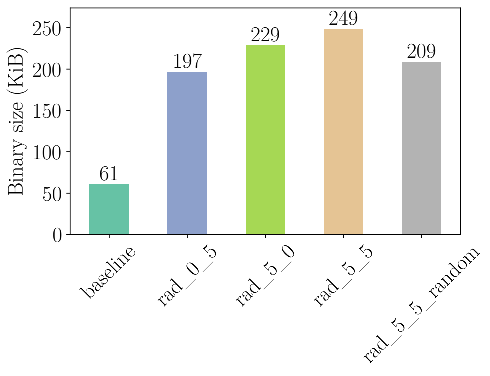
(g) 458.sjeng



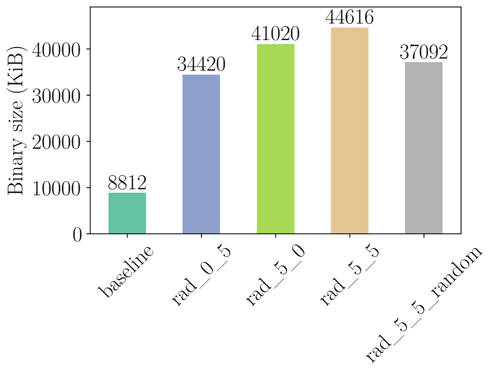
(h) 462.libquantum



(i) 464.h264ref



(j) 473.astar



(k) 483.xalancbmk

Figure A.3: Binary size in KiB of the SPEC CPU2006 benchmarks per benchmark.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliothek, Your knowledge hub
TU WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Tables

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Algorithms

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acronyms

RAD Return Address Decoys. 2, 29–31, 33, 35–39, 43, 46–49, 52, 53, 58, 59, 61–64, 75

VAD VTable Address Decoys. 37–41, 62, 75

W⊕X Write exclusive-or Execute. 14, 17

ABI Application Binary Interface. 6, 7

AOCR Address-Oblivious Code Reuse. 20, 25, 31, 32

ASLR Address space layout randomization. 17, 23, 26, 29, 37

BROP Blind ROP. 19, 35

BSS Block Started by Symbol. 5

BTB Branch Target Buffer. 61

CFG Control-Flow Graph. 17

CFI Control-Flow Integrity. 17, 35

COOP Counterfeit object-oriented programming. 38

COTS Commercial Off The Shelf. 34

CPH Code-Pointer Hiding. 2, 24, 62

CPI Code-Pointer Integrity. 23, 36, 37

DAG Directed Acyclic Graph. 44

GOT Global Offset Table. 64

IR Intermediate Representation. 43

JIT-ROP Just-In-Time ROP. 19, 20, 31, 32, 63, 64

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

LIFO	last in, first out. 4, 7
LTO	Link Time Optimization. 48, 52
MLR	Malicious Loop Reuse. 20
MTB	Malicous Thread Blocking. 20
PC	Program Counter. 61
PLT	Procedure Linkage Table. 64
PSS	Proportional Set Size. 55–57, 75
RISC	Reduced Instruction Set Computer. 9
ROP	Return-Oriented-Programming. 15, 31, 35, 52, 58, 59
RSB	Return Stack Buffer. 11, 37
TBAA	Type Based Alias Analysis. 47
VM	Virtual Machine. 64

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Bibliography

[18] *TIOBE Index | TIOBE - The Software Quality Company.* June 2018. URL: https://www.tiobe.com/tiobe-index/ (visited on 08/22/2019).

[Aba+05] Martín Abadi et al. „A theory of secure control flow". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics).* Vol. 3785 LNCS. 2005, pp. 111–124. ISBN: 3540297979. DOI: 10.1007/11576280_9.

[Aba+09] Martín Abadi et al. „Control-flow integrity principles, implementations, and applications". In: *ACM Transactions on Information and System Security* 13.1 (Oct. 2009), pp. 1–40. ISSN: 10949224. DOI: 10.1145/1609956.1609960. URL: http://portal.acm.org/citation.cfm?doid=1609956.1609960%20http://portal.acm.org/citation.cfm?doid=1102120.1102165.

[Ano01] Anonymous. „Once upon a free". In: (2001). URL: http://phrack.org/issues/57/9.html.

[Bac+14] Michael Backes et al. „You Can Run but You Can't Read". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14.* New York, New York, USA: ACM Press, 2014, pp. 1342–1353. ISBN: 9781450329576. DOI: 10.1145/2660267.2660378.

[BC05] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel.* Third Edit. O'Reilly Media, 2005. ISBN: 978-0596005658.

[Ble+11] Tyler Bletsch et al. „Jump-oriented programming". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security - ASIACCS '11.* New York, New York, USA: ACM Press, 2011, p. 30. ISBN: 9781450305648. DOI: 10.1145/1966913.1966919.

[BSD05] Sandeep Bhatkar, R Sekar, and D.C. DuVarney. „Efficient techniques for comprehensive protection from memory error exploits". In: *Proceedings of the 14th conference on USENIX Security Symposium-Volume 14.* USENIX Association, 2005, pp. 17–17.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[Bur+17] Nathan Burow et al. „Control-Flow Integrity“. In: *ACM Computing Surveys* 50.1 (Apr. 2017), pp. 1–33. ISSN: 03600300. DOI: 10.1145/3054924. URL: http://dx.doi.org/10.1145/0000000.0000000%20http://dl.acm.org/citation.cfm?doid=3058791.3054924.

[Car+15] Nicholas Carlini et al. „Control-Flow Bending: On the Effectiveness of Control-Flow Integrity“. In: *USENIX Security Symposium*. 2015, pp. 161–176. ISBN: 978-1-931971-232.

[Che+10] Stephen Checkoway et al. „Return-oriented programming without returns“. In: *Proceedings of the 17th ACM conference on Computer and communications security - CCS '10*. New York, New York, USA: ACM Press, 2010, p. 559. ISBN: 9781450302456. DOI: 10.1145/1866307.1866370.

[Coh93] Frederick B. Cohen. „Operating system protection through program evolution“. In: *Computers and Security* 12.6 (Oct. 1993), pp. 565–584. ISSN: 01674048. DOI: 10.1016/0167-4048(93)90054-9. URL: https://linkinghub.elsevier.com/retrieve/pii/0167404893900549.

[Con+15] Mauro Conti et al. „Losing Control“. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*. Vol. 2015-Octob. New York, New York, USA: ACM Press, 2015, pp. 952–963. ISBN: 9781450338325. DOI: 10.1145/2810103.2813671.

[Con+16] Mauro Conti et al. „Selfrando: Securing the Tor Browser against De-anonymization Exploits“. In: *Proceedings on Privacy Enhancing Technologies* 2016.4 (Oct. 2016), pp. 454–469. ISSN: 2299-0984. DOI: 10.1515/popets-2016-0050. URL: http://content.sciendo.com/view/journals/popets/2016/4/article-p454.xml.

[Cra+13] Stephen Crane et al. „Booby trapping software“. en. In: *Proceedings of the 2013 workshop on New security paradigms workshop - NSPW '13*. New York, New York, USA: ACM Press, 2013, pp. 95–106. ISBN: 9781450325820. DOI: 10.1145/2535813.2535824.

[Cra+15a] Stephen Crane et al. „It's a TRaP“. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*. New York, New York, USA: ACM Press, 2015, pp. 243–255. ISBN: 9781450338325. DOI: 10.1145/2810103.2813682.

[Cra+15b] Stephen Crane et al. „Readactor: Practical Code Randomization Resilient to Memory Disclosure“. In: *2015 IEEE Symposium on Security and Privacy*. Vol. 2015-July. IEEE, May 2015, pp. 763–780. ISBN: 978-1-4673-6949-7. DOI: 10.1109/SP.2015.52.

[Dav+13] Lucas Davi et al. „Gadge me if you can“. In: *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security - ASIA CCS '13*. New York, New York, USA: ACM Press, May 2013, p. 299. ISBN: 9781450317672. DOI: 10.1145/2484313.2484351.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[Dav+14] Lucas Davi et al. „Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection“. In: *23rd USENIX Security Symposium (USENIX Security 14)*. 2014, pp. 401–416. ISBN: 978-1-931971-15-7.

[Dav+15a] Lucas Davi et al. „HAFIX“. In: *Proceedings of the 52nd Annual Design Automation Conference on - DAC '15*. Vol. 2015-July. New York, New York, USA: ACM Press, 2015, pp. 1–6. ISBN: 9781450335201. DOI: `10.1145/2744769.2744847`.

[Dav+15b] Lucas Davi et al. „Isomeron: Code Randomization Resilient to (Just-In-Time) Return-Oriented Programming“. en. In: *Proceedings 2015 Network and Distributed System Security Symposium*. Reston, VA: Internet Society, 2015. ISBN: 1-891562-38-X. DOI: `10.14722/ndss.2015.23262`.

[Eva+15a] Isaac Evans et al. „Control Jujutsu“. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*. Vol. 2015-Octob. New York, New York, USA: ACM Press, 2015, pp. 901–913. ISBN: 9781450338325. DOI: `10.1145/2810103.2813646`.

[Eva+15b] Isaac Evans et al. „Missing the Point(er): On the Effectiveness of Code Pointer Integrity“. en. In: *2015 IEEE Symposium on Security and Privacy*. Vol. 2015-July. IEEE, May 2015, pp. 781–796. ISBN: 978-1-4673-6949-7. DOI: `10.1109/EuroSP.2016.24`.

[Ge+16] Xinyang Ge et al. „Fine-Grained Control-Flow Integrity for Kernel Software“. In: *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, Mar. 2016, pp. 179–194. ISBN: 978-1-5090-1751-5. DOI: `10.1109/EuroSP.2016.24`.

[Gök+14] Enes Göktaş et al. „Out of Control: Overcoming Control-Flow Integrity“. In: *2014 IEEE Symposium on Security and Privacy*. IEEE, May 2014, pp. 575–589. ISBN: 978-1-4799-4686-0. DOI: `10.1109/SP.2014.43`.

[Gök+16] Enes Göktaş et al. „Undermining Information Hiding (and What to Do about It)“. In: *25th {USENIX} Security Symposium ({USENIX} Security 16)*. 2016, pp. 105–115. ISBN: 978-1-931971-32-4.

[Hom+13] Andrei Homescu et al. „Profile-guided automated software diversity“. In: *Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, Feb. 2013, pp. 1–11. ISBN: 978-1-4673-5525-4. DOI: `10.1109/CGO.2013.6494997`.

[Kil+06] Chongkyung Kil et al. „Address Space Layout Permutation (ASLP): Towards Fine-Grained Randomization of Commodity Software“. In: *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. IEEE, Dec. 2006, pp. 339–348. ISBN: 0-7695-2716-7. DOI: `10.1109/ACSAC.2006.9`.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[KSP14] Volodymyr Kuznetsov, László Szekeres, and Mathias Payer. „Code-pointer integrity". In: *Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation*. Vol. 14. 2014, pp. 147–163. ISBN: 9781931971164.

[Lar+14] Per Larsen et al. „SoK: Automated Software Diversity". In: *2014 IEEE Symposium on Security and Privacy*. IEEE, May 2014, pp. 276–291. ISBN: 978-1-4799-4686-0. DOI: 10.1109/SP.2014.25.

[Mas+15] Ali Jose Mashtizadeh et al. „CCFI". In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS '15*. Vol. 2015-Octob. New York, New York, USA: ACM Press, 2015, pp. 941–951. ISBN: 9781450338325. DOI: 10.1145/2810103.2813676.

[Muc97] Steven Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, 1997. ISBN: 978-1558603202.

[NT14] Ben Niu and Gang Tan. „RockJIT". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*. New York, New York, USA: ACM Press, 2014, pp. 1317–1328. ISBN: 9781450329576. DOI: 10.1145/2660267.2660281.

[One96] Aleph One. „Smashing the Stack for Fun and Profit". In: *Phrack* 7.49 (Nov. 1996). URL: http://www.phrack.com/issues.html?issue=49%7B%5C&%7Did=14.

[Pay12] Mathias Payer. *Too much PIE is bad for performance*. Tech. rep. 2012, p. 3. DOI: 10.3929/ethz-a-007316742. URL: http://phrack.com/issues.html?issue=67%7B%5C&%7Did=.

[Rud+17] Robert Rudd et al. „Address Oblivious Code Reuse: On the Effectiveness of Leakage-Resilient Diversity". In: *Proceedings 2017 Network and Distributed System Security Symposium*. Reston, VA: Internet Society, 2017. ISBN: 1-891562-46-0. DOI: 10.14722/ndss.2017.23477.

[Sch+15] 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*. Vol. 2015-July. IEEE, May 2015, pp. 745–762. ISBN: 978-1-4673-6949-7. DOI: 10.1109/SP.2015.51.

[Sha+04] Hovav Shacham et al. „On the effectiveness of address-space randomization". In: *Proceedings of the 11th ACM conference on Computer and communications security - CCS '04*. New York, New York, USA: ACM Press, 2004, p. 298. ISBN: 1581139616. DOI: 10.1145/1030083.1030124.

[Sha07] Hovav Shacham. „The geometry of innocent flesh on the bone". In: *Proceedings of the 14th ACM conference on Computer and communications security - CCS '07*. Vol. 22. 4. New York, New York, USA: ACM Press, 2007, p. 552. ISBN: 9781595937032. DOI: 10.1145/1315245.1315313.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.

[Sno+13] Kevin Z. Snow et al. „Just-in-time code reuse: On the effectiveness of fine-grained address space layout randomization“. In: *Proceedings - IEEE Symposium on Security and Privacy*. IEEE, May 2013, pp. 574–588. ISBN: 9780769549774. DOI: `10.1109/SP.2013.45`.

[Sta11] William Stallings. *Operating Systems: Internals and Design Principles*. Pearson Learning Solutions, 2011. ISBN: 9780132309981.

[Sze+13] László Szekeres et al. „SoK: Eternal War in Memory“. In: *2013 IEEE Symposium on Security and Privacy*. IEEE, May 2013, pp. 48–62. ISBN: 978-0-7695-4977-4. DOI: `10.1109/SP.2013.13`.

[Tic+14] Caroline Tice et al. „Enforcing Forward-Edge Control-Flow Integrity in {GCC} & {LLVM}“. In: *23rd {USENIX} Security Symposium ({USENIX} Security 14)*. San Diego, CA: {USENIX} Association, Aug. 2014, pp. 941–955. ISBN: 978-1-931971-15-7.

[TSS15] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. „Heisenbyte“. In: *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security - CCS ’15*. Vol. 2015-Octob. New York, New York, USA: ACM Press, 2015, pp. 256–267. ISBN: 9781450338325. DOI: `10.1145/2810103.2813685`.

[War+12] Richard Wartell et al. „Binary stirring“. In: *Proceedings of the 2012 ACM conference on Computer and communications security - CCS ’12*. New York, New York, USA: ACM Press, 2012, p. 157. ISBN: 9781450316514. DOI: `10.1145/2382196.2382216`.

[Wer+16] Jan Werner et al. „No-Execute-After-Read“. In: *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security - ASIA CCS ’16*. New York, New York, USA: ACM Press, 2016, pp. 35–46. ISBN: 9781450342339. DOI: `10.1145/2897845.2897891`.

[Zha+16] Chao Zhang et al. „VTrust: Regaining Trust on Virtual Calls“. In: *Proceedings 2016 Network and Distributed System Security Symposium*. Reston, VA: Internet Society, 2016. ISBN: 1-891562-41-X. DOI: `10.14722/ndss.2016.23164`.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar.
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub