Multi-Architecture Binary Rewriter to Prevent ROP Arbitrary Code Execution

MATTEO PIANO

Master of science in Software Engineering
Date: January 18, 2018
KTH Supervisor: Roberto Guanciale
KTH Examiner: Mads Dam
PoliTo Supervisor: Maurizio Rebaudengo

KTH - School of Computer Science and Communication
Politecnico di Torino - Dipartimento di Automatica e Informatica

Abstract

Despite the increasing attention to the topic of computer security, the amount of vulnerable software services is still high. The exploitation of a common vulnerability like memory management bugs brought to the development of an attack known as *Return Oriented Programming* (ROP). Such technique employs malicious memory injections to hijack the control flow of the targeted application and execute an arbitrary series of instructions.

This thesis explores the design and implementation of a static binary rewriting tool able to instrument applications compiled for the Linux operating system in order to offer protection against ROP exploitation on x86 and ARM platforms. The instrumentation is achieved by extracting re-compilable assembler code from executable binary files which is then processed and modified. The effectiveness of such solution is tested with a selection of benchmarking utilities in order to evaluate the cost in terms of performance caused by its employment. The results obtained from these experiments show that on average the added overheads are acceptably low and, consequently, the proposed tool is a valid solution to improve the security of vulnerable applications when the original source code is not available.

Sammanfattning

Trots den ökande uppmärksamheten på ämnet datasäkerhet är mängden sårbara mjukvarutjänster fortfarande stor. Utnyttjandet av en vanlig sårbarhet som minneshanteringsfel har lett till utvecklingen av en attack som kallas *Return Oriented Programming* (ROP). Denna teknik utnyttjar skadliga minnesinjektioner för att ändra kontrollflödet för den riktade applikationen och utföra en godtycklig serie instruktioner.

Detta exjobb undersöker utformningen och genomförandet av ett verktyg för statisk binär omskrivning som kan användas för att instrumentera applikationer för Linux-operativsystemet för att erbjuda skydd mot ROP-exploatering på x86- och ARM-plattformar. Instrumentering uppnås genom att extrahera återkompilerbar assemblerkod från exekverbara binära filer som sedan behandlas och modifieras. Effektiviteten av sådan lösning testas med ett urval av benchmarkingverktyg för att utvärdera kostnaden när det gäller prestanda som orsakas av dess användning. Resultaten från dessa experiment visar att de extra kostnaderna i genomsnitt är acceptabelt låga och, följaktligen, är det föreslagna verktyget en giltig lösning för att förbättra säkerheten för sårbara applikationer när den ursprungliga källkoden inte är tillgänglig.

Acknowledgments

I sincerely want to thank professor Roberto Guanciale for providing the indications which led to the definition of this project and for actively supervising my work during the whole development process.

A great thank goes my family which supported me throughout my studies and gave me the opportunity to experience university life also beyond the borders of my home country.

Finally, I feel the need to also thank the StackOverflow community for helping me solve many programming issues during the years.

Contents

Chapter 1

Introduction

1.1 Project area

It is an undeniable fact that in recent years it has been possible to observe a rising awareness in both the private and the public sector around the subject of computer security. However, this did not really change the fact that the amount of vulnerable services is still very high and, considering the immense growth currently experienced by the software industry, probably increasing.

A common category of vulnerability often found in applications is the incorrect implementation of memory management. The exploitation of such defects can allow a malicious user to inject arbitrary data in the program's memory space, effectively manipulating its behavior during runtime. In order to achieve this result, the injected payload may contain either the machine encoding of an additional code section to be inserted within the original one or plain numeric values designed to hijack the executable's control flow. The first kind has already been addressed and solved with the adoption of memory protection mechanisms. These are supported by all the major modern operating systems, e.g. *Data Execution Prevention* (DEP) in Microsoft Windows and the *NX flag* in the Linux kernel, and implement the *Write XOR Execute* policy ($W \oplus X$) which consists in preventing executable segments from being overwritten and writable segments from being executed. Nevertheless, these strategies are ineffective in contrasting the second type of injection since the exploited machine code is already part of the legitimate binary file.

A particular flavor of this family of attacks is denominated *Return*

Oriented Programming (ROP) [1]. Such technique consists in modifying the control flow of a program by "poisoning" the memory locations containing the target address of indirect branches (returns from procedures, jumps/calls using pointers). This modification allows the attacker to chain a series of code fragments (*gadgets*) in order to perform a desired operation.

This thesis project investigates a static solution able to solve this type of issue by modifying the machine code of existing executable binary files compiled for different computing architectures.

1.2 Problem statement

The subject of the project is the development a tool able to statically analyze and rewrite executable binary files in order to make these secure against ROP attacks and evaluate the performance penalty caused by such instrumentation. The work is mainly based on the research performed by Onarlioglu et al. [2] and wants to extend it by eliminating the need for original source code and by designing similar solutions for architectures different from the already analyzed Intel's *x86-32*, most importantly its 64 bit variant (*x86-64*) and the *Advanced RISC Machine* (ARM) 32 bit *Thumb* instruction set.

The mentioned ROP countermeasure *G-Free* [2] was originally designed to act as preprocessor to the `GNU Assembler` to sanitize programs at compile time from any exploitable *gadget*. It is tailored for the *x86-32* architecture and is shown to deliver good results regarding both security and performance. Its more evident limitations are the need of the program's original source code, since this means requiring software distributors themselves to use a patched compatible compiler in order to benefit from the tool, and the applicability to only one system architecture. The extension of this concept as a binary rewriting tool able to act on different instruction sets allows a greater flexibility and, at the same time, maintains the positive aspects of the original prototype.

The implementation of the project is aimed to work on the *Linux* operating system [3] and, consequently, the rewriting task is performed on *Executable and Linkable Format* (ELF) binary files [4].

The empirical evaluation of the obtained results is conducted by applying the developed tool to a suite of benchmarking software in order to measure the average performance penalty introduced by such pro-

cess. The effectiveness of the approach is also shown by demonstrating how a simple program containing memory management vulnerabilities is instrumented and protected against possible attempts of ROP exploitation.

1.3 Outline

This document presents in Chapter 2 a complete background to the subject matter with details about the targeted processor architectures, an historical overview of the research performed on *Return Oriented Programming* attacks and the state of the art in the fields of binary instrumentation and ROP protection mechanisms. Chapter 3 then discusses the design decisions and implementation strategies employed in the development of the presented tool. Chapter 4 and Chapter 5 contain a description of the quantitative results obtained and final considerations about the effectiveness and relevance of the research work which was carried out.

Chapter 2

Background

The chapter provides an overview of the various subjects relevant in the scope of the project, from architectural aspects to a lineup of ROP attack and defense mechanisms.

2.1 Technical details

2.1.1 x86

The term x86 refers to a family of instruction set architectures (ISAs) based on 8086 CPU produced by Intel in 1978. It was developed with a *complex instruction set computer* (CISC) design and was originally intended to work with 16 bit long registers. Over the years, with the advancements in the areas of microprocessor research and manufacturing, several extensions were made for it by Intel and other manufactures (e.g. AMD) in order to accommodate the introduction of new 32 and 64 bit processors, yet maintaining backwards compatibility.

The design of x86 uses instruction encodings with variable length (from 1 to 15 bytes), hence privileging code density over regularity. It supports single byte memory addressing and employs the little-endian (the least significant byte at the lowest address) convention in managing multi-byte values.

2.1.1.1 Registers

The modern x86 architecture presents 8 general purpose registers (GPRs). These still carry the 8086 naming scheme in 16 bits operations and use

the prefixes "E-" and "R-" when addressing respectively 32 and 64 bits. They are:

- an accumulator register (AX) and a data register (DX) for arithmetic operations;
- a base register (BX) to be employed as data pointer;
- a counter register (CX) to perform shift operations and manage loops;
- a stack pointer register (SP) which stores the address to the top of the stack;
- a stack base pointer register (BP) used to browse values on the stack;
- a source index register (SI) and a destination index register (DI) for stream operations.

If considering a 64 bit CPU, other 8 registers need to be added to this list with names from R8 to R15 and no particular usage convention. Moreover, when dealing with floating operations, current x86 floating point units (FPUs) have a stack with 8 more dedicated registers (named from ST0 to ST7) and, if *Streaming SIMD Extensions* (SSE) is supported, another 8 "large" 128 bit registers (named from XMM0 to XMM7). In addition to the GPRs, there are 6 segment registers which store the addresses of the various memory sections (e.g. stack, code, data) of the current process, though modern operating systems tend to disable their use in favor of a paging methodology. Finally, the two special registers IP and FLAGS respectively contain the address of the next instruction to be executed and a collection of bits representing the resulting state of the previous operation.

2.1.1.2 Procedure calling conventions

Procedures (or subroutines, functions) are computational units executing a certain task. These are invoked by means of the `call` instruction which stores the current IP value on the stack and performs a jump to the target address, which can be either a constant value or a variable saved in a register or in memory. Subroutines ends with `ret` instruction which pops the return address from the stack copying it back to IP.

The x86 architecture presents different conventions when handling the invocation of subroutines.

For 32 bit systems, the one employed most commonly by modern compilers is the so called *cdecl* convention. In it, subroutine arguments are passed on the stack and the return value is stored either in EAX if integer or in ST0 if floating point. The task of preserving the original value of the registers is left to the callee (the procedure itself), with the exception of EAX, ECX and EDX. The preamble of a function usually contains the code necessary to create a new stack frame and allocate space for local variables while its epilogue performs the opposite actions, releasing stack allocation and restoring the old frame.

```
func:
  push EBP            ; store previous frame pointer
  mov  EBP, ESP       ; copy new frame pointer
  sub  ESP, var_size  ; allocate space for local variables
  ...                 ; retrieve arguments from stack
  ...                 ; and do calculations
  mov  EAX, result    ; copy return value
  mov  ESP, EBP       ; free stack allocation
  pop  EBP            ; restore EBP value
  ret                 ; jump back to caller
```

Listing 2.1: x86 32 bit function convention

The main difference introduced in the calling convention for 64 bit systems [5] is that parameter passing is done using registers. RDI, RSI, RDX, RCX, R8 and R9 are employed for the first six integer arguments while the XMM registers are used for floating point operands. However, the stack is still used when a greater number of values is required to be passed.

2.1.2 ARM

The *Advanced RISC Machine* (previously *Acorn RISC Machine*) (ARM) is a family of *reduced instruction set computer* (RISC) architectures primarily developed towards the use in mobile systems and embedded devices. Similarly to the x86 architecture, also ARM chips exist in 32 and 64 bit flavours and, as many other RISC architectures, they are designed with fixed length instructions (more on this in Section 2.1.2.1) and a large number of general purpose registers. These operate following a load-store paradigm which means that operands are always loaded

from memory to registers before executing operations.

2.1.2.1 Instruction sets

The standard instruction set for 32 bit ARM microprocessor is *ARM32* which encodes every instruction in 4 bytes and can make use of all the available features of the underlying CPU. Since this design choice suffers from the fact of having a quite low code density, most processors also have support (and in some particular cases only have support) for another instruction set denominated *Thumb*. The latest version of *Thumb* (*Thumb2*) uses a mixture of 16 and 32 bit instructions which form a subset of the regular *ARM32* instruction set and achieve a quite large increase in code density though loosing some features (e.g. conditional execution [6]). The 64 bit versions of the ARM architecture abandoned this concept with the *AArch64* instruction set which prefers the regularity and capabilities of fixed length 32 bit instructions.

2.1.2.2 Registers

The 32 bit version of the ARM ISA is designed with 16 GPRs named from R0 to R15. The three registers with the highest numberings (R13, R14 and R15) respectively also cover the roles of stack pointer (SP), link register (LR) and program counter (PC). Furthermore, there also exists the *current program status register* (CPSR) which, like FLAGS in the x86 architecture, is a bit collection reflecting the current state of execution. It also contains the so called T field which indicate whether the processor is currently interpreting instructions as *Thumb*. In order to handle floating point operations, 16 double precision registers (from D0 to D15) are present and can also be treated as 32 single precision registers (from S0 to S31).

The systems supporting the *AArch64* ISA have a significantly increased quantity of registers with 31 GPRs (from X0 to X30) and a separate dedicated register for the program counter. Also, the number of floating point registers is doubled compared to the 32 bit version.

2.1.2.3 Procedure calling conventions

In the ARM architecture subroutines are invoked using the *branch and link* (`bl`) operator which stores the current value of the program counter in LR and executes a jump to the indicated target. The arguments are

passed using the first 4 GPRs and return values are store in R0 and R1. The stack is employed in case more than 4 parameters are necessary. The function prologue usually contains the code to store the return address on the stack and create a new stack frame (R11 and R7 are used as frame pointer respectively in *ARM32* and *Thumb* mode) while the epilogue presents the inverse operations. Since none of the 32 bit instruction sets contains a return instruction, the same result is achieved directly copying the value of LR in PC.

```
func:
  push  {r7, lr}      ; save frame pointer and return address
  sub   sp, var_size  ; allocate space for local variables
  add   r7, sp, offs  ; set frame pointer
  ...                 ; do calculations
  mov   r0, result    ; copy return value
  adds  r7, offs
  mov   sp, r7        ; free stack allocation
  pop   {r7, pc}      ; restore frame pointer and return
```

Listing 2.2: ARM32 function convention

The calling convention for *AArch64* is slightly dissimilar given the different composition of registers. The arguments are passed with the first 8 registers (from X0 to X7), X30 is the link register holding the return address and X29 is used as the frame pointer. Yet another difference can be found in the fact that a proper return instruction to copy X30 to the program counter exists.

```
func:
  stp   x29, x30, [sp,#-offs]! ; store frame pointer and
                               ; return address while
                               ; allocating local space
  mov   x29, sp                ; set frame pointer
  ...                          ; do calculations
  mov   w0, result             ; copy return value
  ldp   x29, x30, [sp], #offs  ; restore frame pointer and
                               ; return address while
                               ; freing stack allocation
  ret                          ; return
```

Listing 2.3: AArch64 function convention

2.2 Return oriented programming

This section provides an overview of the evolution over time of the exploitation techniques which led to the development of the attack category named *Return Oriented Programming*.

2.2.1 Buffer overflow

Citing the words of the notorious computer scientist Edsger Dijkstra, "testing can only reveal the presence of errors, never their absence". Recent times have seen an increase in sensibility towards software security and, consequently, a growth in the investments in software validation. Nevertheless, applications designed employing lower level programming languages, which leave to the programmer himself or herself the final decision on memory management, are often prone to present defects in this area. The issue of badly administered memory buffers has almost always been among the most frequently reported software vulnerabilities and with the highest severity scores [7].

In simple words, buffer overflows mostly occur when a section of code reads some kind of input and copies it into memory without performing the necessary boundary checks. If the size of the input data is greater than the one of the destination buffer, the copying process exceeds the buffer boundaries and the contiguous memory locations are overwritten. Accordingly to how the buffer was created, this memory violation may happen either in the *heap* (section used for dynamic memory allocation) or in the *stack* (section used for static allocation, local variables, return addresses). For the subject matter, the latter case is the one of greater interest since it allows an attacker to modify the values of local variables and return addresses hence hijacking the control flow of the running process. Appendix A.1.1 and Appendix A.1.2 show simple examples of stack overflow exploitations in C programs.

2.2.2 Return-to-lib(c)

Once a buffer overflow attack is performed and the control flow has been hijacked, the execution can be directed to different sections of the process. The so called *Return-to-lib(c)* attacks have been one of the first milestones towards modern ROP exploits.

The first known proof of this concept was rendered publicly available by Designer in 1997 [8]. The technique consists in inserting on the stack the address of a library function in the place of a return address and modelling the values in the other positions nearby to be the desired arguments for such routine. In this way, when reaching the next return instruction the program would jump to and execute the function in the way chosen by the attacker. A common choice as target function is the `system()` system call which creates a child process that executes a shell command by means of which the attacker can operate further violations.

2.2.3 Return oriented programming on x86

Despite their power, *return-to-libc* exploits are often quite complicated to execute given the difficulty in locating memory addresses of library procedures and the variations in parameter passing. This last detail became particularly relevant with the introduction of 64 bit architectures together with new procedure calling conventions where most of the parameters are passed using registers [5].

Further research led to the work published in 2007 by Shacham [9] which overcame these limitations by removing the need of invoking library functions. He presents the idea of *Return Oriented Programming* (ROP) and the concept of *gadget*, a portion of machine code with an indirect branch, for example a return instruction, at its end. Moreover, taking advantage of the variable size of the x86 instruction set, he demonstrates how *gadgets* can not only be found in function prologues but also be constructed by means of unaligned execution as shown in Figure 2.1. He asserts the result of chaining a reasonably sized series of *gadgets*, as before by corrupting the values of addresses stored on the stack, to compose a Turing-complete system hence able in principle to perform any allowed operation on the underlying architecture and solve any computational problem a Turing machine could tackle.

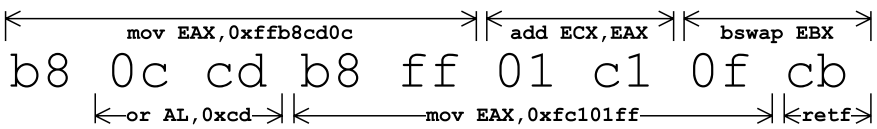

Figure 2.1: Aligned and misaligned interpretation of x86 machine code

2.2.4 Return oriented programming on RISC

As a follow up to Shacham's publication, new groundwork was made to apply his conception to system architectures different from x86, in particular to *reduced instruction set computer* (RISC) architectures (e.g. ARM, SPARC, MIPS). A characteristic shared by most of the exponents of this category is the use of fixed length instruction encodings with enforcement for aligned execution, in contrast to the natural code density of Intel's binaries. Furthermore, these often present procedure calling and return conventions different from the ones of the x86 architecture.

In 2008 Buchanan et al. presented an adaptation of ROP to the SPARC architecture [10]. They describe the different challenges imposed by the structure of such instruction set and design a *gadget* catalog for memory-to-memory operations from the Solaris standard C library composing a Turing-complete system.

Likewise, in 2010 Kornau introduced in his thesis [11] the applicability of ROP on the ARM architecture, also describing an algorithm to determine whether a code library contains the necessary amount of *gadgets* in order to craft any arbitrary program.

2.3 Binary instrumentation

Binary instrumentation is the process used to insert new instructions in an existing executable file in order monitor its operations or introduce changes in its behavior. There exist two different approaches to this technique: it can be performed dynamically at runtime by directly processing the program code when loaded into memory or statically by operating the desired modifications on the binary file.

The first method benefits from having available those pieces of information which can be obtained only after an executable is loaded and dynamically linked, though often leading to a quite important performance degradation and a high memory usage. Some examples of dynamic instrumentation frameworks are DynamoRIO [12], Dyninst [13], Pin [14] and Valgrind [15].

The latter has on average a much smaller impact on resource usage during execution (it depends on the type of instrumentation applied), but it is sometimes limited in its applicability due to the non-trivial effort of retrieving and interpreting all the necessary data from the binary file. This kind of instrumentation is usually achieved either

by altering function entry points to jump to a different code section where the function code is copied and modified or by translating the binary to an intermediate representation (it may also be assembler language), applying the desired revisions and finally recompiling it to an executable format. Some known frameworks are Dyninst [16] (different API of the same tool mentioned above), PEBIL [17], SecondWrite [18] and Uroboros [19].

2.4 Related work

Different solutions to address the issue of *Return Oriented Programming* have been proposed over the years.

2.4.1 Monitoring

A naive observative solution specifically designed to detect ROP attacks is accomplished by dynamically performing at runtime a frequency analysis of the executed instructions, searching for those commonly present when going through a chain of *gadgets* (e.g. `ret`) [20, 21]. This approach has its field of applicability but can be defeated when variations to the attack methodology [22] are employed and the frequency analysis fails to recognize them.

Other specific ROP solutions utilize a *Control Flow Graph* (CFG) to verify the control flow of the program [23, 24]. A CFG is a preventively computed graph of all the possible execution patterns of the application at issue where nodes are the various code sections and the edges indicate control flow transfers. After its evaluation, the CFG is then used to instrument the program in such way that the integrity of return addresses and jump pointers is controlled and a failure occurs in case its control flow does not follow the allowed directions described in the graph. These techniques tend to cause on average a fairly high performance overhead for the resulting binary and usually to require a long analysis time. In a similar manner, other solutions employ the calculated flow graph to dynamically check the validity of the current state of the program, making them applicable for remote attestation of embedded systems [25, 26].

2.4.2 Stack protection

More generic tools make an attempt to solve the problem by trying to detect memory violations (e.g. buffer overflows) and avoid any sort of data injection, hence eliminating the starting cause of the vulnerability.

A possible approach to stack protection is the creation of a so called *shadow stack* in a different memory segment. This is used to store a copy of the return addresses and serves as a reference to prove their correctness when needed. An example of this type of mechanisms is ROPdefender [27] which accomplishes it by means of dynamic binary instrumentation. Despite its good suitability against basic ROP attacks, the large number of runtime checks causes it to suffer a quite large negative impact on performance.

Another strategy designed to the purpose of stack protection, like for example in the compiler extension StackGuard [28], is the use of *canary* values. These are known data words which are pushed onto the stack preceding control flow related values and act as a proof of their integrity since a buffer overflow attack, in order to reach the targeted location, would modify the value of the *canary* as well. The main defect of such approach is that it relies on the assumption that the *canary* is always verified before the execution of an indirect branch, which may not be the case if an adversary manages to make the program start executing instructions in a misaligned manner.

In this regard, *G-Free* [2] is another compiler based solution which improves stack protection effectiveness by systematically modifying the assembler source code of the program in order to eliminate the instructions which could be interpreted as indirect branches by an unaligned execution. These instructions are denominated *free branches* and include returns from procedures and indirect jumps and function calls which use addresses stored in memory. As previously mentioned, the next chapters discuss how the concept introduced in this last methodology is used as starting point to create a static binary rewriter to protect x86 and ARM applications from ROP attacks and other free branch exploitations.

2.4.3 Address Space Layout Randomization

Address Space Layout Randomization (ASLR) [29] is a security feature introduced in most modern operating systems which randomizes the position of the different memory segments (stack, heap, code, etc.)

inside the address space of a process. This makes it difficult for attackers to design the right shell-code to successfully mount an attack since memory offsets need to be guessed. This mechanism would offer a quite high level of security but often its implementation is excessively naive and uses permutations with too low entropy, which leaves it vulnerable to brute force attacks and other kinds of offensives [30].

Chapter 3

Methods

This chapter provides a description of the techniques employed in the development of the project. It firstly presents the underlying concept of the framework used to make binary instrumentation possible, then it discusses the modifications which are applied to the targeted binaries in order to protect them against *Return Oriented Programming* attacks.

3.1 Executable reconstruction

As previously discussed in Section 2.3, there exist different approaches when addressing the problem of the modification of existing executable files. The technique which was chosen in the context of this work tackles the concern by de-constructing the original binary to the corresponding processor instructions and, performing an analysis in search of some common compilation patterns, extracting an assembler source file which can then be processed by the GNU C compiler.

The original concept for this technique was presented in the prototype designed by Wang et al. named *Uroboros* [19]. Such tool was developed exploiting powerful type system offered by the OCaml language in conjunction with standard Unix command line utilities like `readelf` and `objdump`. Their design undertakes the reconstruction of x86 executables (targeting both 32 bit and 64 bit architectures) and, to pursue the intent of this project, it consequently had to be extended to also support the processing of ARM 32 bit *Thumb* binaries. In order to smoothen the aforementioned operation, the entire prototype has been rebuilt employing a more flexible and familiar language such as Python with the aid of some reverse engineering libraries (e.g. Cap-

stone [31]) and, at the same time, a trimming of some functionalities unnecessary for the subject matter has been operated allowing slightly faster performances.

The main steps which are adopted in order to perform the reconstruction of assembly source code are the following:

1. the various sections of the ELF binary file (e.g. `.text`, `.rodata`, `.got`,...) are separated and the section containing the executable code (`.text`) is processed by a disassembler which recovers machine code in readable ASCII text format;

2. the code is parsed and transcoded to a small internal symbol system in order to facilitate subsequent operations;

3. the intermediate representation is scanned searching for constant values which may represent pointers to data or other positions of the code, replacing these with labels;

4. a similar process is applied to the data sections using a word-sized (4 bytes for 32 bit executables, 8 bytes for 64 bit executables) scanning window;

5. the newly generated labels are added in the locations of the disassembled code and data sections corresponding to their original addresses;

6. information about external functions (from dynamically linked libraries) and global variables (e.g. standard IO file handles) are extracted from the ELF header and the corresponding symbols inserted in the code;

7. the code and data are dumped back to a source text file which can be compiled again to an executable.

The next subsections describe some architecture-dependent transformations which are carried out by the tool in order to correctly handle some particular constructions introduced by compilers in binary files.

3.1.1 x86

The most important aspect to address when processing x86 binaries is the transformation of position independent code (PIC). Are included in

this category all those instructions which make use of relative offsets from the current location (the value of the program counter) to access other addresses of the executable's memory space. This characteristic makes the code agnostic of the virtual address is loaded at and, consequently, suitable for libraries and for other scenarios where code positions may be varied, like ASLR [29].

The 64 bit version of x86 manages this mechanism in a simple and straightforward manner. The compiler uses the program counter (RIP), which contains the address of the next instruction to be executed, as base register for "register plus offset" indirect addressing expressions, like shown in Listing 3.1.

```
4005C9: mov RAX, [RIP + 0x200858] ; 4005D0 + 200858 = 600E28
4005D0: ...
...
600E28: .quad some_value
```

Listing 3.1: x86 64 bit PC relative addressing

In order to preserve the correct references, after the decompilation phase, the tool searches for instructions containing this pattern and substitutes the relative offset with the correct absolute address. This allows such constants to be recognised as addresses during the analysis phase and substituted by labels, from which the compiler is then able to re-evaluate the appropriate offsets for the reassembled executable. The assembler syntax for this operation may result misleading, since the destination label is directly summed to the instruction pointer, but this kind of indirect memory addressing is recognised by compilers which substitute to the label its offset from the current position instead of its absolute value.

```
mov RAX, [RIP + S_0x600E28]
...
S_0x600e28: .quad some_value
```

Listing 3.2: x86 64 bit PC relative addressing after processing

On the other hand, the 32 bit version of the instruction set does not support direct use of the instruction pointer register in PC relative addressing. Listing 3.3 shows how the value of the program counter is retrieved by exploiting the operations performed by the `call` instruction: a stub `thunk` procedure is invoked and the return address

is extracted from the stack and stored in a register. Moreover, offsets are calculated not from the current position but from the address of the _GLOBAL_OFFSET_TABLE_ (a symbol in the .got.plt section), which increases the difficulty in correctly recovering the memory references.

```
get_pc_thunk.bx:
8049210: mov  EBX, [ESP]    ; store return address from stack
8049213: ret
...
806D697: call get_pc_thunk.bx
806D69C: add  EBX, 0x26964 ; 806D69C + 26964 = 8094000
...
806D6C0: mov  EAX, [EBX + 0x12300] ; 8094000 + 12300
...                                ; = 80A6300
8094000: _GLOBAL_OFFSET_TABLE_
...
80A6300: .word some_value
```

Listing 3.3: x86 32 bit PC relative addressing

The solution adopted to correctly interpret this construction scans the disassembled code looking for thunk function invocations and subsequent indirect addressing expressions where the register in which the instruction pointer was loaded is used as a base, then substituting them with the absolute value of the pointed address. The assumption which is made by this approach is that such base register is only valid within the boundaries of the function where the thunk routine is called. Even though indirect addressing is substituted by absolute values, the invocation to the thunk function and the calculation of the address of the _GLOBAL_OFFSET_TABLE_ are maintained in the tentative of not breaking indirect references which eventually were not recognised during the scanning process.

```
call get_pc_thunk.bx
add  EBX, OFFSET _GLOBAL_OFFSET_TABLE_
...
mov  EAX, [S_0x80A6300]
...
S_0x80A6300: .word some_value
```

Listing 3.4: x86 32 bit PC relative addressing after processing

3.1.2 ARM

The stricter constraints imposed by the RISC design of the ARM architecture require compilers to adopt particular strategies in the composition of executable programs. As a consequence, in order to correctly recover re-compilable assembler source code, such patterns must be recognised and transformed.

3.1.2.1 PC relative load

One of the biggest limiting factors of the *Thumb* instruction set is the limited bit length (usually no more than 8/12 bits) usable to represent constant values. This has an important impact on the use of both absolute and relative addressing since it becomes difficult to handle addresses in large executable files. One of the most employed solutions to this issue is the use of "in-line data" embedded in the `.text` section in-between executable instructions and accessed by means of PC relative loads without the need of large offsets (must be smaller than 4096 bytes). It must be noted that also in *Thumb* mode the program counter is always incremented by 4 and, consequently, like in Listing 3.5, the value to be used for calculating the source address is the 4-byte aligned virtual address of the instruction increased by 4 ($pc = CurrAddr - (CurrAddr \ mod \ 4) + 4$).

```
10556: ldr    r6, [pc, #0x2C] ; 10558 + 2C = 10584
...
10584: .word some_value
```

Listing 3.5: ARM PC relative load

Since these data values are found in the code section, they must be handled and transformed during the disassembling phase in order not to erroneously interpret them as instructions. This is achieved by simply collecting the targets of PC relative load instructions, skipping their targets and substituting the "PC plus offset" expression with a constant value of the source address which is then translated to a label. The byte-size of the skipped binary data is inferred analyzing the utilized variant of the `ldr` operator.

```
ldr r6, S_0x10584
...
S_0x10584: .word some_value
```

Listing 3.6: ARM PC relative load after processing

There also exist a variant of `ldr` operation which can be used to load from memory a double word (64 bit) value over two registers. In such cases the usable value range for offset is even smaller hence causing compilers to prefer to apply the offset with a separate `add` instruction with a 12 bit immediate value (this operation is also aliased as `adr`).

```
27FD6: add  r1, pc, #0x640 ; 27FD8 + 640 = 28618
27FDA: ldrd r0, r1, [r1]
...
28618: .word some_value
2861C: .word some_other_value
```

Listing 3.7: ARM large offset PC relative load

Similarly to the previous case, these instructions are handled during the decompilation phase by calculating the targeted address and performing appropriate substitutions.

```
adr  r1, S_0x28618
ldrd r0, r1, [r1]
...
S_0x28618: .word some_value
           .word some_other_value
```

Listing 3.8: ARM large offset PC relative load after processing

When even this approach is not sufficient, mainly in cases where the data cannot be located in the `.text` section (e.g. when writable) and as a consequence very large offsets are required in order to reach such memory locations with PC relative operations, compilers may choose to store the offset itself as "in-line data" and then loading it in a register which is added to the program counter. It must be noted that, when the program counter register appears as second source operand in addition in *Thumb* mode, its value is calculated as the address of the current instruction increases by 4.

```
18128: ldr r7, [pc, #0x284] ; 18128 + 284 = 183AC
...
1814C: add r7, pc           ; r7 = 1814C + 4 + A338 = 22488
...
18154: ldr r1, [r7]
...
183AC: .word 0xA338
...
22488: .word some_value
```

Listing 3.9: ARM very large offset PC relative load

The biggest issue in handling this kind of situation is that, as underlined in Listing 3.9, the involved instructions are often not grouped together, probably for optimization reasons, which causes this pattern to be hard to recognise. If successfully individuated, such procedure is transformed like show in Listing 3.10.

```
ldr r7, S_0x183AC
...
add r7, #0
...
ldr r1, [r7]
...
S_0x183AC: .word S_0x22488
...
S_0x22488: .word some_value
```

Listing 3.10: ARM very large offset PC relative load after processing

3.1.2.2 Jump tables

The strategy employed by compilers to build jump tables is another example of data values embedded in the code section of the executable hence requiring special care during the disassembling phase. The *Thumb* instruction set supports two different flavours of such structure: absolute address tables and offset tables.

The first kind is built using the `ldr` operator to copy a new value in the program counter register thus triggering a branch. As shown in Listing 3.11, the employed instructions simply evaluate an address corresponding to a table entry from an index and then load its value.

```
...                             ; evaluate index
217BE: cmp   r3, #0x5A          ; compare index with table size
217C0: bhi   0x2198C            ; jump to default if higher
217C4: add   r2, pc, #0x4       ; r2 = 217C8 + 4 = 217CC
217C6: ldr   pc, [r2, r3, lsl #2] ; load table[r3]
217CA: nop                          ; to program counter
217CC: .word 0x2197F
217D0: .word 0x2198D
...
21938: other instructions
```

Listing 3.11: ARM absolute jump table

The loading procedure is the same which was previously introduced when dealing with PC relative loads with large offset. In this case the separate addition operation used to calculate the table address is due to the fact that the program counter register cannot appear in both operands of the `ldr` instruction. The other difference is that in this context the size of the data "gap" cannot be inferred from the load instruction itself but must evaluated from the comparison operation performed on the index register, whose immediate operand represents the number of table entries (the byte size is equal to the table length multiplied by 4). Keeping in consideration these details, that applied transformation are the same used for normal "in-line data" and the resulting from such processing is shown in Listing 3.12.

```
cmp   r3, #0x5A
bhi   S_0x2198C
adr   r2, S_0x217CC
ldr   pc, [r2, r3, lsl #2]
nop
S_0x217CC: .word S_0X2197F
           .word S_0X2198D
           ...
S_0x21938: other instructions
```

Listing 3.12: ARM absolute jump table after processing

The second type of jump table makes use of either the `tbb` or the `tbh` operator. These accept as argument an indirect memory addressing pointing to an entry in the offset table and perform a relative jump whose length is the double of the read offset. The only difference between these instructions is the size of the loaded offset, a single byte and two bytes respectively.

```
...                             ; evaluate index
11EDE: cmp     r3, #0x6         ; compare index with table size
11EE0: bhi     0x121B6          ; jump to default if higher
11EE4: tbh     [pc, r3, lsl #1] ; jump to PC + 2 * table[r3]
11EE8: .short 0x0160
11EEA: .short 0x0090
...
11EF6: other instructions
```

Listing 3.13: ARM offset jump table

In order to correctly preserve offset values, these have to be transformed in label expressions, as shown in Listing 3.14, by calculating the absolute branch destination, subtracting the base address of the jump table and dividing the result by 2. The amount of data to retain from disassembling is evaluated like in the previous case by looking for comparison instructions on the index register.

```
cmp     r3, #0x6
bhi     S_0x121B6
tbh     [pc, r3, lsl #1]
S_0x11EE8: .short (S_0x121A8 - S_0x11EE8) / 2
           .short (S_0x12008 - S_0x11EE8) / 2
           ...
S_0x11EF6: other instructions
```

Listing 3.14: ARM offset jump table after processing

3.1.2.3 Double move

Another technique exploited by compilers to load large values into registers is splitting these in two 16 bit parts a loading them using two `mov` operations (the `movt` operand can load values in the upper 16 bits of a register). In the process of binary disassembly and reconstruction this is of particular importance when such value represents a pointer and, consequently, has to be replaced by a label in order to maintain correct memory references. Listings 3.15 and 3.16 show an example of such pattern and how it is transformed using special assembler directives after the analysis process.

```
14124: movw r0, #0x102C
14128: movt r0, #0x2
...
2102C: something
```

Listing 3.15: ARM double mov

```
movw r0, #:lower16:S_0x2102C
movt r0, #:upper16:S_0x2102C
...
S_0x2102C: something
```

Listing 3.16: ARM double mov after processing

A particular which had to be considered very carefully in such analysis is that the pair of `mov` operations is often not contiguous and, in some cases, the register in which the lower 16 bits are stored can be copied to another register (or even on the stack) on which the upper 16 bits are then moved.

3.2 Instrumentation

This section discusses the instrumentation process applied to the disassembled code in order to offer protection against *Return Oriented Programming* attacks. The concepts here introduced were originally designed by Onarlioglu et al. for the 32 bit x86 architecture in their *G-Free* prototype [2] and were adapted for the other architectures of interest in this project. The basic idea is to modify the code of an application in order to eliminate the possibility of performing indirect branches exploiting unaligned code interpretation and at the same time protecting legal *free branch* instructions from being misused.

3.2.1 Aligned execution enforcement

As previously enunciated in Section 2.2.3, one of the most powerful features of ROP is the use of unaligned execution to construct a Turing complete set of *gadgets*. One of the purposes of the proposed instrumentation strategy is to try to force aligned execution by means of code transformation and insertions.

3.2.1.1 x86

The variable length and density of the x86 instruction set cause this architecture to be particularly sensible to unaligned code execution. The main purpose of this portion of the instrumentation is to ensure that, when bytes which encode indirect branches (e.g. `0xc2`, `0xc3`, `0xca`, `0xcb` for the `ret` instruction) are present in other instructions, these cannot be exploited for control flow hijacking.

The main solution adopted towards solving this problem is the insertion of *alignment sleds* before the instructions containing such critical bytes. A *sled* is a long enough sequence of instructions without effects which forces the program execution to preserve its alignment. The easiest way to implement such structure is to employ a series of `nop` instructions which, being encoded in a single byte (`0x90`), allows to easily manage the length of the *sled*. Also, a jump instruction is prepended to the *sled* so that when execution is correctly aligned the impact on performance is minimal. Figure 3.1 shows how a possible misaligned *gadget* is sanitized using this technique. When undesired bytes are found in the least significant bits of a jump offset, a shorter *sled* (just 1 or 2 bytes) can be employed before or after such instruction in order to modify the code layout and cause a change of the offset such that it will not contain *free branch* encodings anymore.

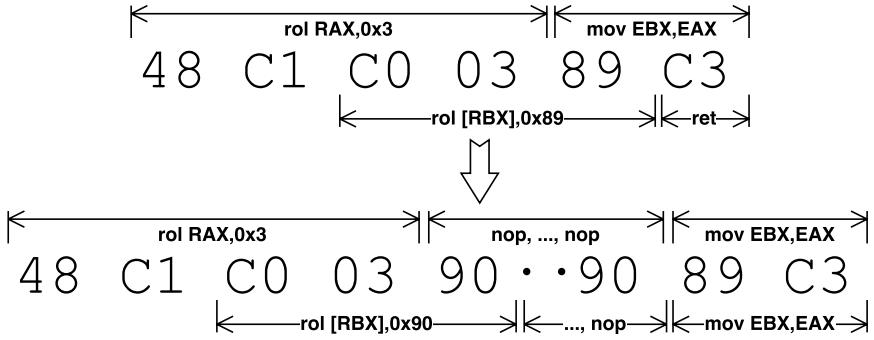

Figure 3.1: Example of alignment sled application in x86 code

Another solution to avoid undesired byte encodings is the replacement or transformation of certain instructions. A simple example is `movnti` (`0x0f 0xc3`) which copies a value from memory minimizing cache pollution and can be safely replaced by a normal `mov` instruction. Other code transformations include the modification of immediate operands containing *free branch* opcodes. In most cases these can be

split in multiple instructions performing the same operation like shown in Listing 3.17.

```
sub RAX, 0xC3      ->    sub RAX, 0xC4
                         inc RAX
```

Listing 3.17: Immediate value transformation

The original *G-Free* prototype, being integrated in the compiler toolchain, also performed modifications in register allocation in order to avoid undesired bytes in the `ModR/M` and `SIB` fields x86 instruction. This is obviously very difficult to achieve in the context of direct binary modification and such cases are handled using *alignment sleds* instead at the price of some loss in performance.

3.2.1.2 ARM

The issue of unaligned interpretation is not as relevant when taking in consideration the execution of *Thumb* code since the architecture itself enforces a 2 byte alignment. It is however possible that, in presence of a code block containing multiple consecutive 32 bit long instructions, a branch could be directed to such block with 16 bits of offset thus making the processor execute the last 2 byte of an instruction together with the first 2 of the successive one (of course, only in the case those first 2 byte encode the beginning of a 32 bit instruction). This can be avoided by inserting a short ineffective 2 byte instructions (e.g. `nop` or `mov r0,r0`) in order to "break" the block of 32 bit instructions (Figure 3.2) and ensure that, even though a misaligned branch is performed, the execution will return to normal after at most one unintended instruction.

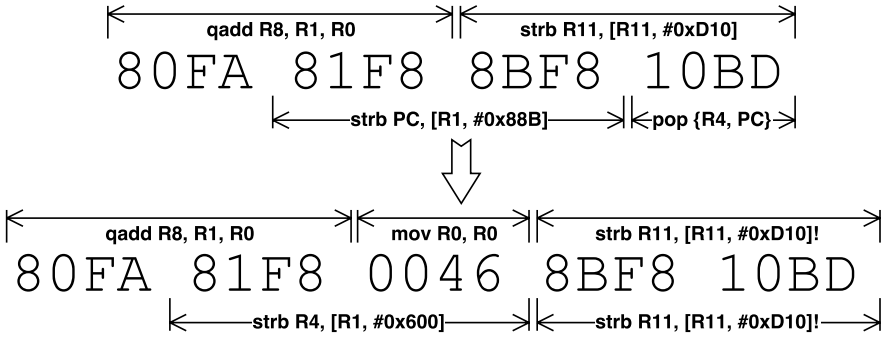

Figure 3.2: Example of alignment enforcement in Thumb code

3.2.2 Return address protection

The successive step after constraining the execution of the program to only follow intended aligned code is to protect legitimate return instructions in order to prevent adversaries from using them for *gadget* chaining. This is achieved by implementing XOR canaries at the entry and exit point of each function. In other words, this kind of instrumentation adds at the beginning of functions a small header which uses a previously generated key to encrypt the value of the return address present on the stack. Likewise, another short piece of code is added before the return statement which uses the same key to decrypt the return address to its original form. In such manner, if an attacker tries to modify the control flow in a way which does not respect the correct function entry points, the decryption will generate a non-controllable incorrect address, which will most likely result in a crash for the application. The XOR operator is chosen as the preferred cryptographic tool because of its high computational efficiency and its quality of preserving the probability distribution of the pseudo-random encryption key in the encrypted data. The encryption key is generated by a routine invoked at the beginning of the program's execution and stored in memory (example code in Appendix A.2). Listings 3.18 and 3.19 provide an overview of instrumentation code employed for the targeted architectures (for x86 the 32 and 64 bit instrumentations are merged in a single listing given the fact that only register prefixes vary).

```
func:                                   ; function entry
  push (E|R)AX                          ; save AX
  mov  (E|R)AX, [xorkey]                ; load key
  xor  [(E|R)SP + 0x(4|8)], (E|R)AX     ; encrypt return address
  pop  (E|R)AX                          ; restore AX
  ...                                   ; function body
  push (E|R)AX                          ; save AX
  mov  (E|R)AX, [xorkey]                ; load key
  xor  [(E|R)SP + 0x(4|8)], (E|R)AX     ; decrypt return address
  pop  (E|R)AX                          ; restore AX
  ret                                   ; return
```

Listing 3.18: x86 (32|64) bit XOR canary

As mentioned in Section 2.1.1.2, the ARM instruction set does not have a built-in stack-based procedure calling and it is the compiler's duty to generate instructions which move the link register (LR) on and

off the stack in order to create call frames. This particularity facilitates the encryption process since the return address is already loaded on a register and can be easily managed. More attention must instead be paid when modifying exit points since the return address is usually directly moved from the stack to the program counter register. In order to allow the decryption operation, the `pop` operation is modified to move the encrypted value back to the link register which is then decrypted and used to branch back to the caller.

```
func:                           ; function entry
  push {r0}                     ; save r0
  movw r0, #:lower16:xorkey     ; load key address
  movt r0, #:upper16:xorkey
  ldr  r0, [r0]                 ; load key
  eor  lr, r0                   ; encrypt return address
  pop  {r0}                     ; restore r0
  push {..., lr}                ; save return address to stack
  ...                           ; function body
  pop  {..., pc -> lr}          ; (original exit point)
                                ; get return address from stack
  push {r0}                     ; save r0
  movw r0, #:lower16:xorkey     ; load key address
  movt r0, #:upper16:xorkey
  ldr  r0, [r0]                 ; load key
  eor  lr, r0                   ; decrypt return address
  pop  {r0}                     ; restore r0
  bx   lr                       ; return
```

Listing 3.19: ARM XOR canary

3.2.3 Indirect branch protection

The other employed defense mechanism serves the purpose of protecting jumps or procedure calls which have a variable value (from a register or a memory location) as destination. To this end, in the functions which contain such instructions, additional header and footer are added in order to ensure that the program execution is respecting the function's entry and exit points. This is achieved by using a *frame cookie* generated at the beginning of the function and checked before every indirect branch.

The original *G-Free* implementation stores such cookie on the program's stack, thus modifying its layout, and then performs an in depth

code analysis in order to correct any stack reference which is rendered invalid by the presence of this additional memory block. This solution, though also acting as an indicator of stack integrity, resulted of difficult application, especially when compiler optimizations have been utilized and make stack management hard to track. For this reason, the presented *frame cookie* implementation makes use of a separate dedicated stack created in the *Thread Local Storage* (TLS) memory area.

The value of the cookie is generated by applying the XOR operation on a random key (different from the one employed in return address encryption) and a unique identifier assigned to the function at issue and is then combined with the counter representing the current size of the *cookie stack* in order to make it different for multiple invocations of the same function. Before any indirect jump or call, the cookie is retrieved and this operation is reversed in order to check its validity and, if not present or not valid, the application is terminated with an error. Of course, at the end of the instrumented functions the cookie is removed from the stack.

Moreover, in the case of ARM binaries, the instrumentation also introduces a check on the register holding the destination address of the indirect branch to make sure that this is not employed to change the running mode from Thumb to standard ARM32 (ARM processors utilize the least significant bit of branch addresses to determine the running mode to be employed after the jump, 0 for ARM32 and 1 for Thumb). Since such interworking is necessary when branching to the ARM32 encoded entries in the `.plt` section used in the invocation of linked library functions, first the register value is checked to represent an address inside the `.text` section and then, if the check succeeds, an OR operation is used to forcefully set the last bit of the address to 1.

Due to their length, code snippets of the assembler implementations of the presented techniques are moved in Appendix A.3.

3.2.4 Further adjustments for ARM executables

Some of the characteristics of the ARM architecture impose quite strict constraints in the structure of source code. The machine code produced by compilers for mid/high level programming often barely fits in such constraints and, consequently, the introduction of supplementary instrumentation instructions causes some code structures to break. The following list presents the cases which had to be addressed in order to

be able to produce compilable programs after the previously described processing.

- The instructions `cbz` and `cbnz` are employed to perform short jumps when the value of the register operand is respectively zero and non-zero. These are limited only to branches with positive offsets lying between 4 and 130 bytes. These limits are very restrictive and can be easily violated when inserting new code. In such cases, these instructions are transformed in a standard zero comparison and a conditional jump.

```
cbz r0, dest    ->    cmp r0, #0
...                   beq dest
...                   ...
dest: ...             dest: ...
```

Listing 3.20: ARM cbz transformation

- A similar issue of out of range offset presents itself with the PC relative `vldr`. This instruction is used to load a floating point value located with an offset from -1020 to +1020 bytes from the current code location (this is usually a data value embedded in the code section). For the same reasons stated in the previous point, this has to be transformed in a series of instructions employing a temporary register to hold the data address.

```
vldr s0, val    ->    push {r0}
...                   movw r0, #:lower16:val
...                   movt r0, #:upper16:val
...                   vldr s0, [r0]
...                   pop  {r0}
...                   ...
val: .word ...        val: .word ...
```

Listing 3.21: ARM vldr transformation

  It theoretically also exists the corresponding store instruction `vstr` but, in practice, this is never found in executables since writable memory section are always too far away from the code section.

- Another adjustment made necessary by the increased code size regards jump tables using byte-long offsets with the `tbb` instruction.

When the displacement values cannot be contained in a single byte, these require to be transformed using the `tbh` operator.

```
tbb [pc, r1]          ->    tbh [pc, r1, lsl #1]
tab:                        tab:
 .byte (dest-tab)/2          .short (dest-tab)/2
...                         ...
dest: ...                   dest: ...
```

Listing 3.22: ARM tbb transformation

- A particular feature of the ARM architecture is the possibility of conditional execution of instructions different from branches. This is achieved with the use of `it` blocks, which mark the conditional code portion, and operator suffixes, which indicate the type of condition. These code blocks represent a problem when alignment sleds need to be inserted between their components and, consequently, are translated using normal conditional branches when the instrumentation is applied.

```
ite   eq          ->    beq it_true
addeq r0, #1            sub r0, #1
subne r0, #1            b   it_cont
...                     it_true: add r0, #1
...                     it_cont: ...
```

Listing 3.23: ARM it block transformation

Chapter 4

Results

The following sections show how the instrumentation process impacts the performance of the targeted executable and provide an assessment of enhanced security of a processed binary.

4.1 Performance analysis

4.1.1 Benchmarks

The performance evaluation was operated by applying the instrumentation process on a small selection of benchmarks from the open source *Phoronix Test Suite* [32]. The following list describes which tests were employed and how they were configured. All the experiments were performed on the latest available version of the softwares at the time of testing (October 2017).

- **blake2s**: *BLAKE2* [33] is a cryptographic hash function. The benchmark based on it (provided by the authors themselves in the *BLAKE2* repository) consists in multiple runs of the function on different message sizes. The test was build on the reference sequential version of the function in order to allow compilation for multiple architectures (there exist versions employing specialized instructions to improve performance).
- **gzip-compress**: *GNU zip* is a file compression utility based on the DEFLATE algorithm [34, 35]. The benchmark consists in compressing a 2GB randomly generated file.

- **dcraw**: *dcraw* is a command line utility able to decode various formats of RAW digital photographs. The benchmark consists in converting two images, of 10Mp and 12Mp resolution, from Nikon's RAW file format (NEF) to *Portable PixMap* format (PPM).
- **encode-flac**: *Free Lossless Audio Codec* (FLAC) is, as its name says, a lossless audio format. The benchmark consists in converting a 16 bit PCM 44100Hz WAV file of 7 minutes and 44 seconds (the 10th track from the CCA licensed album *The Slip* by *Nine Inch Nails*) with the best quality.
- **himeno**: *Himeno Benchmark* is a test developed by Ryutaro Himeno. Since the benchmark is designed to always run in about 60 seconds, the reported results are normalized values which represent the time necessary to complete 1000 iterations (the test reports the total number of internal iterations performed).
- **bzip2-decompress**: *bzip2* is a file compression utility based on the Burrows–Wheeler algorithm [36]. The benchmark consists in decompressing an archive containing the version 3.7 of the Linux kernel.
- **dolfyn**: *dolfyn* is an open source *Computational Fluid Dynamics* (CFD) utility developed in Fortran. The benchmark consists in running the tool on the `demo` dataset distributed together with its source code.
- **encode-opus**: *Opus* [37] is an open source lossy audio codec. The benchmark has the exact same structure of **encode-flac**.

The testing machine employed for x86 benchmarks is a HP Pavilion G6 notebook [38] with a third generation Intel Core i5 3230m processor, 8GB of DDR3-1600 system memory and a 480GB SanDisk Ultra II solid state drive [39] running Linux Mint 18.2 Sonya on the version 4.11.0-14-generic of the Linux kernel. ARM testing was instead performed on a dedicated server with a Marvell Armada XP quad-core SoC at 1.33GHz [40], 2GB of system memory and 50GB of SSD storage running Ubuntu 16.04 on the version 4.4.95-mainline of the Linux kernel. The compilation of the executables was handled by the version 5.4 of the GNU compiler tool-chain (both for C and Fortran) with level 2 optimizations. Execution times were recorded using the `real` output of *Bourne Again*

SHell (Bash) built-in `time` function with millisecond precision. The shown results are calculated as the average of three best executions cycles sampled from sets of 10 or more in the attempt of smoothen the influence of other running processes and system interrupts. All the benchmarking results are available in Appendix B.1.

4.1.2 Reconstruction

Figures 4.1 to 4.3 show the relative overhead introduced by the reconstruction process alone, calculated as

$$RelativeOverhead = \left(\frac{ReassembledTime}{OriginalTime} - 1 \right) \cdot 100 \, .$$

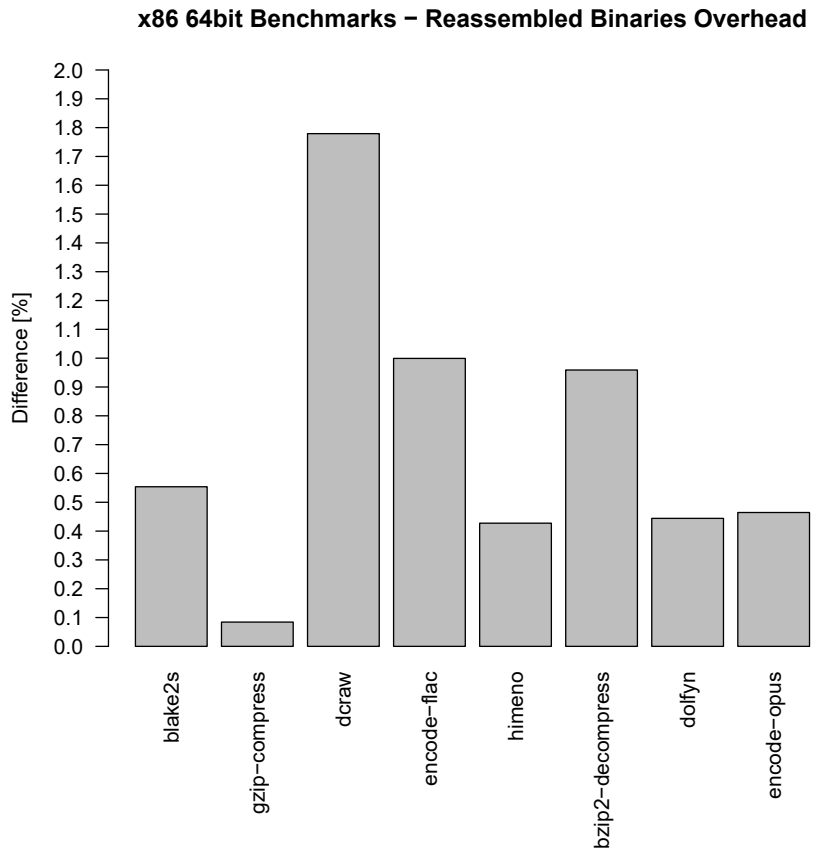

Figure 4.1: Benchmark overheads (average) for reassembled x86 64 bit executables

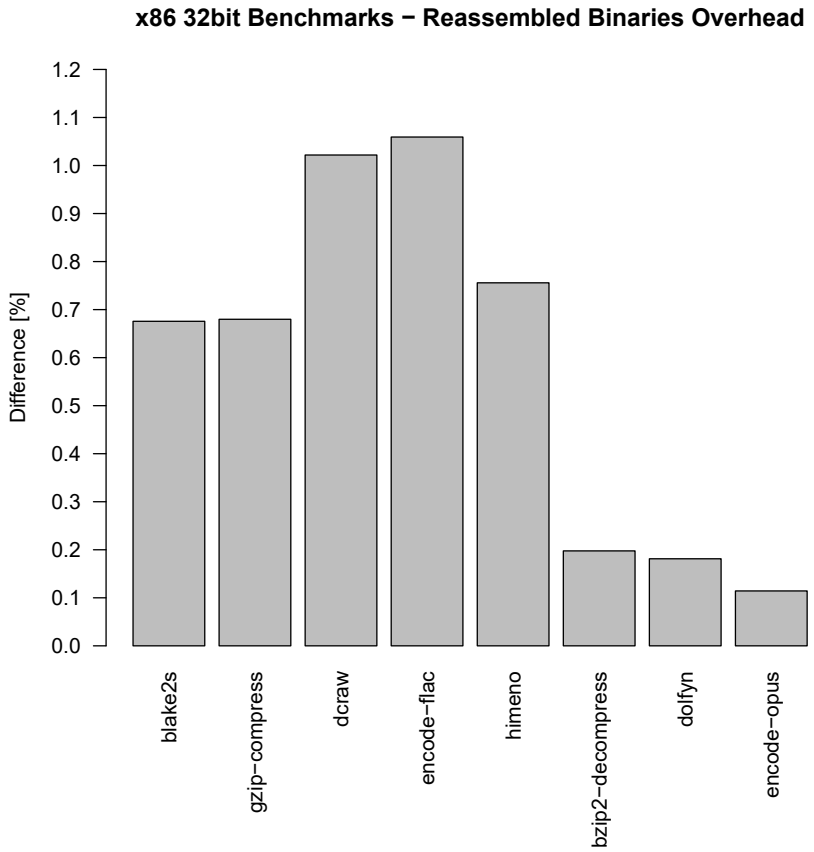

Figure 4.2: Benchmark overheads (average) for reassembled x86 32 bit executables

It can be seen that the disassembly and reassembly routine has a really low (lower than 1%) impact on the performance of the targeted binaries. These results are in line with what has already been discussed in the paper about the original *Uroboros* implementation [19]. The cause for the slight changes in execution speed can most probably be found in differences in memory accesses and instruction fetching. The reconstruction procedure, as explained in the previous chapter, must apply some modification to memory addressing in order to maintain correct references and, furthermore, it also has the tendency to generate executables with small differences in code and data alignment.

Given its slightly higher overhead on the x86 64 bit architecture, further investigations were done on the **dcraw** benchmark using *Linux* `perf` *tool* [41] which, although only producing approximate results, can

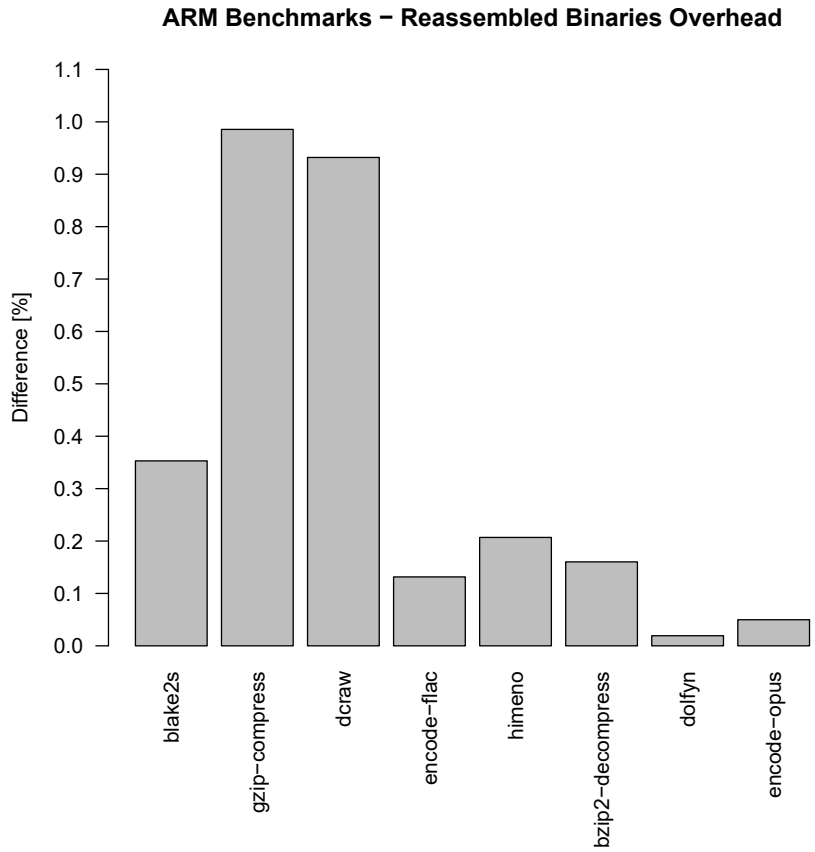

Figure 4.3: Benchmark overheads (average) for reassembled ARM executables

provide good process profiling. The tool highlighted a 10% increase in branch mis-prediction during the execution of the reassembled binary which can be labelled as the most probable cause for speed penalty.

4.1.3 Instrumentation

Figures 4.4 to 4.6 show the relative overhead recorded on the execution of the binaries processed by the instrumentation tool, calculated with the same formula shown in the previous section.

The performance penalty for x86 binaries in almost all cases hovers below 10%/12%. Small executables like **blake2s** and **himeno**, having a much lower number of functions and function invocations, tend to present an even lower impact on speed. The *Linux* `perf` *tool* was

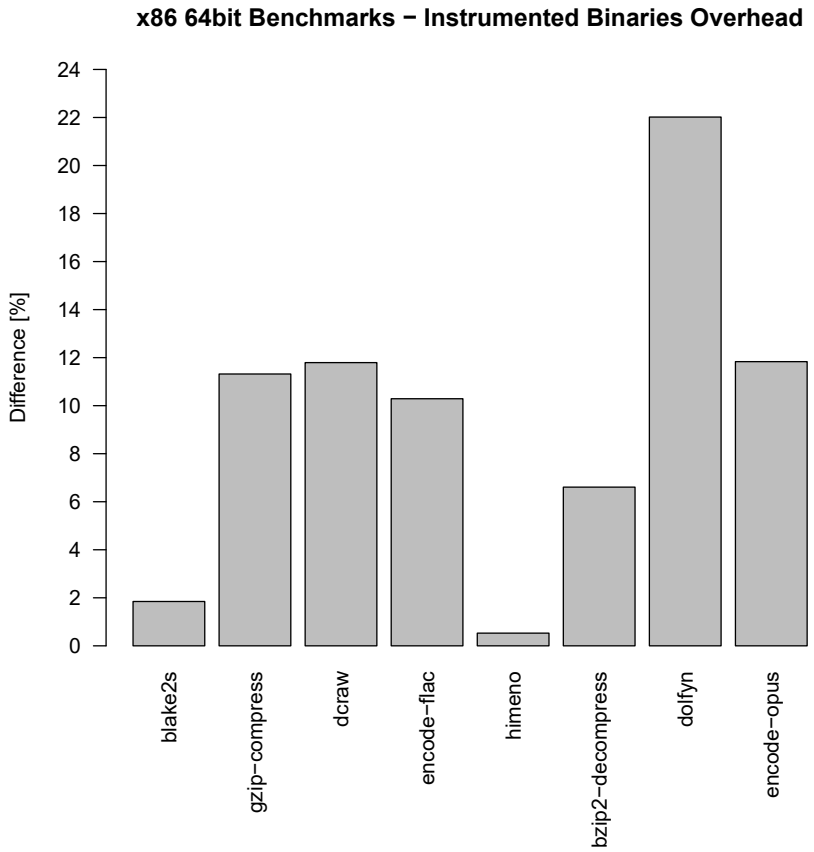

Figure 4.4: Benchmark overheads (average) for instrumented x86 64 bit executables

again used to have a better understanding of the anomalous-looking results. In the 64 bit benchmarks, **dolfyn** suffers a 21% greater amount of cache misses which, given the very short overall execution time of the program, results in a quite significant relative speed overhead. The instrumented 32 bit version of **dcraw** presents 32% more branch mis-predictions and an increase of 49% in the number of stalled CPU cycles caused by memory accesses. This could be traced back to the significantly large number of indirect `jmp/call` instructions present in the program which cause several accesses (for some reason more expensive in 32 bit mode) to the thread local storage to manage *frame cookies*. Also, probably due to the bottleneck introduced by disk read throughput, it can be noted that the performance difference for the 32 bit **gzip** benchmark is very low. This same situation does not repeat

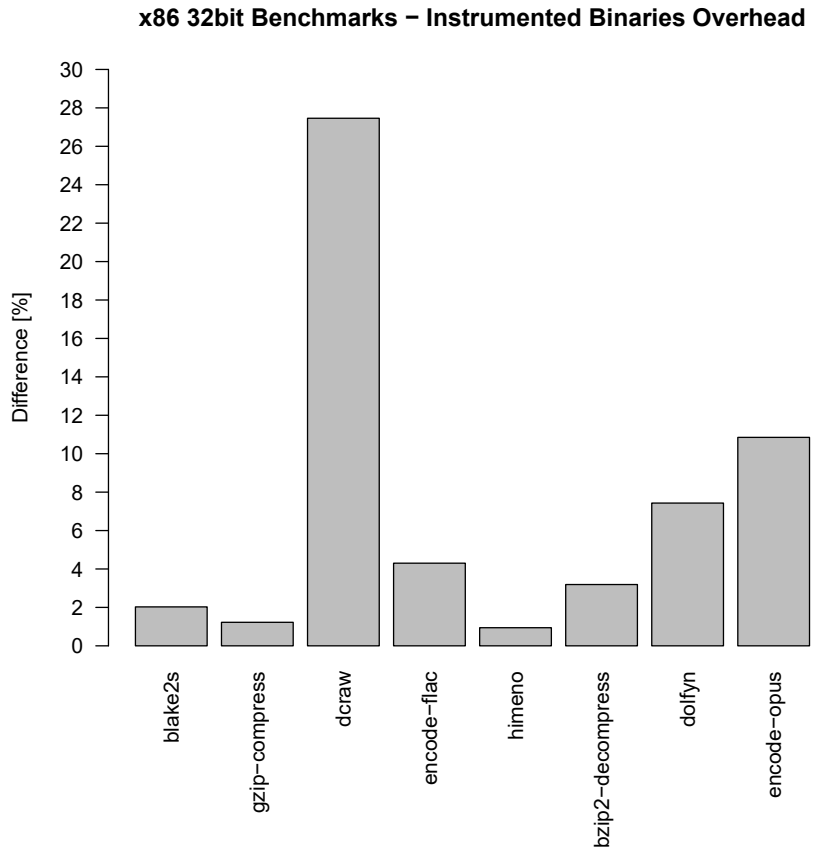

Figure 4.5: Benchmark overheads (average) for instrumented x86 32 bit executables

for the 64 bit version probably because of the 80% higher branch misprediction recorded by `perf`.

Taking in consideration the results from benchmarking instrumented ARM binaries, some of the recorded values appear to be anomalous. The majority of tests present very low overhead or, in cases like **dolfyn** and **bzip2**, even a speed improvement. The fact of this outcome being highly unlikely due to the increased number of instructions to be executed led to further investigations. These indicated that the performance bias was caused by different execution efficiency between the instruction blocks "`pop {pc}`" and "`pop {lr}; bx lr`". As explained in Section 3.2.2, this transformation is applied during the instrumentation process in order to allow return address decryption and from standalone testing a speed difference between 48% and 51%

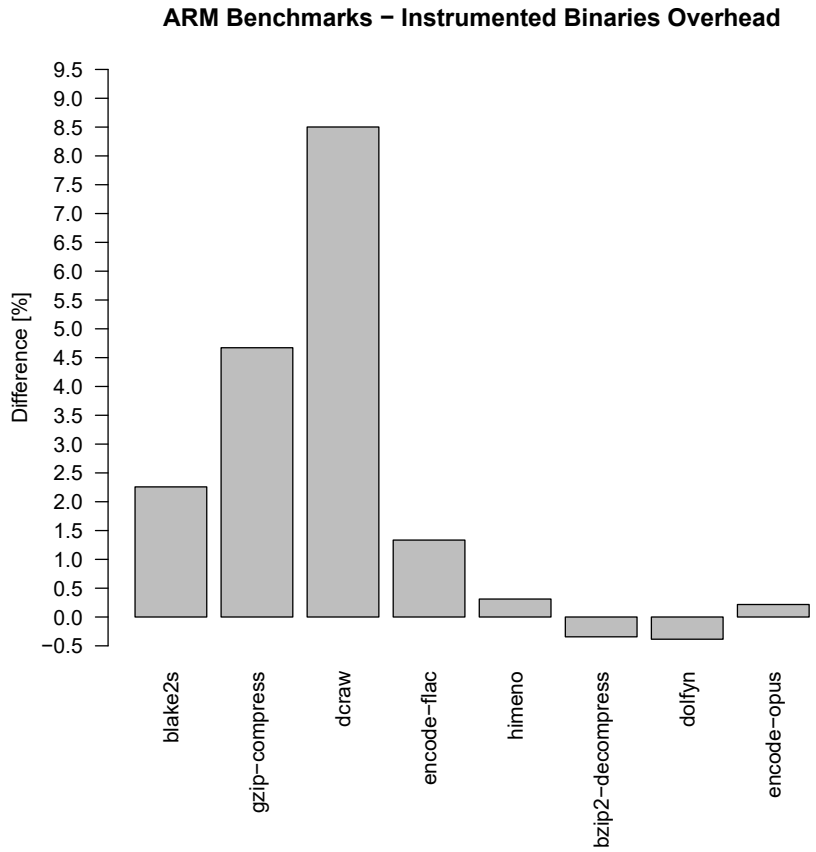

Figure 4.6: Benchmark overheads (average) for instrumented ARM executables

emerged in favour of the latter form. The testing was also repeated on a *BBC micro:bit* [42] board running an ARMv6 Cortex-M0 processor [43] and, since no significant variation between the code fragments was evidenced on such system, the drawn conclusion was to attribute the performance abnormality to the ARMv7 implementation of the testing system.

Nevertheless, **dcraw**, **gzip** and **blake2s** present marginally higher overheads and the `perf` analysis reported an increase of more than 50% in branch mis-prediction for the first two and a 4 times higher number of cache misses for the latter.

4.2 File sizes

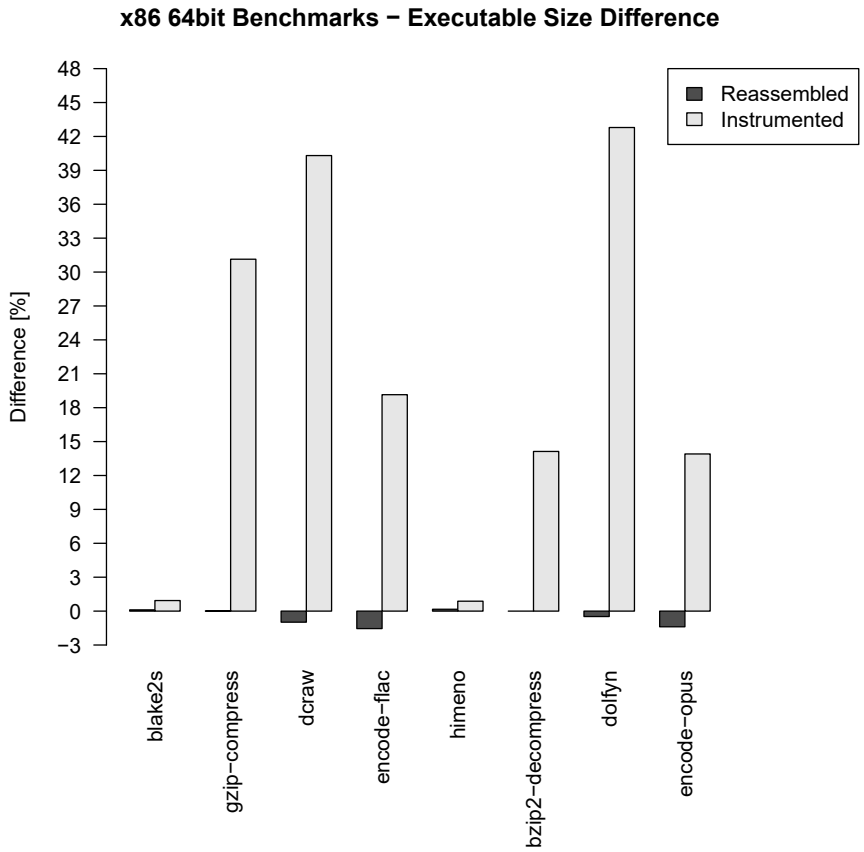

Figure 4.7: Benchmark executable file size difference for x86 64 bit binaries

Figures 4.7 to 4.9 show the relative file size variation caused by the reassembly and instrumentation processes. The measurements were operated on executables with all symbol information removed in order to take into account only the size increase produced by the added code and data pieces. The charts show very low differences for the reassembled binaries and an added size between 15% and 45% for the instrumented binaries, approximately in proportion to the number of functions of the executable. Instrumented ARM binaries present on average a lower size overhead than the x86 ones due to the less frequent adoption of *alignment sleds*.

The negative size variations recorded for most of the reassembled

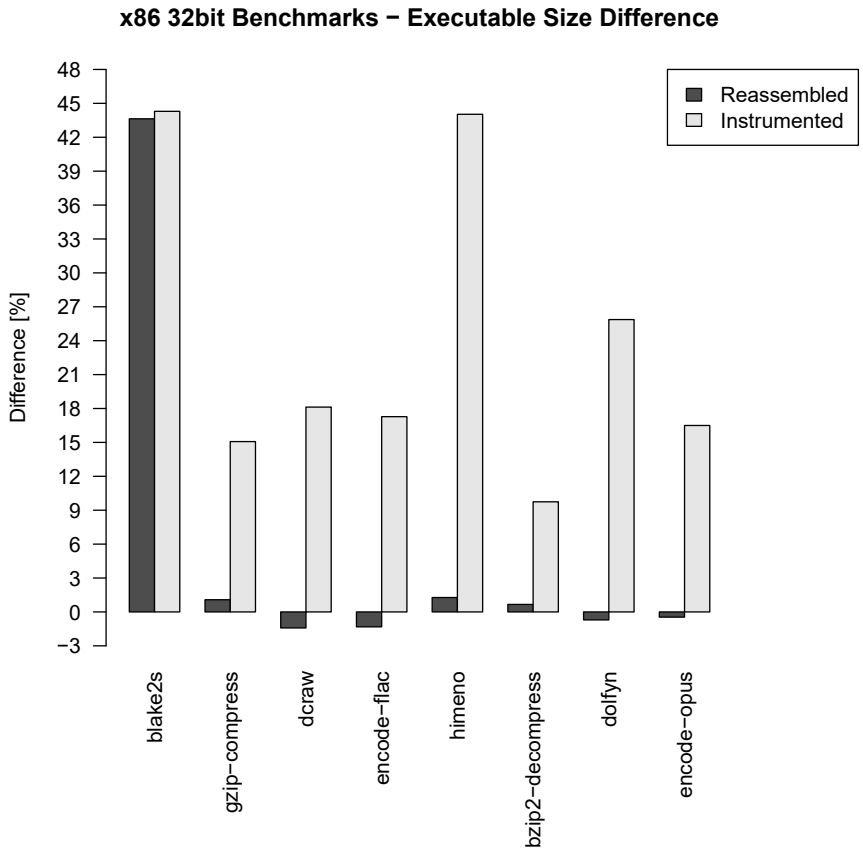

Figure 4.8: Benchmark executable file size difference for x86 32 bit binaries

x86 binaries are due to padding instruction (e.g. `lea ESI, [ESI + EIZ * 1 + 0x0]`), which can be generated with extended length by higher level compilers, to be recompiled using standard encoding hence resulting in shorter byte sequences.

The very high relative values displayed for the x86 32 bit versions of **blake2s** and **himeno** are caused by the fact that the code sections of the original binaries are small enough to be contained in a single 4KB page while the applied processing makes the addition of another memory page required.

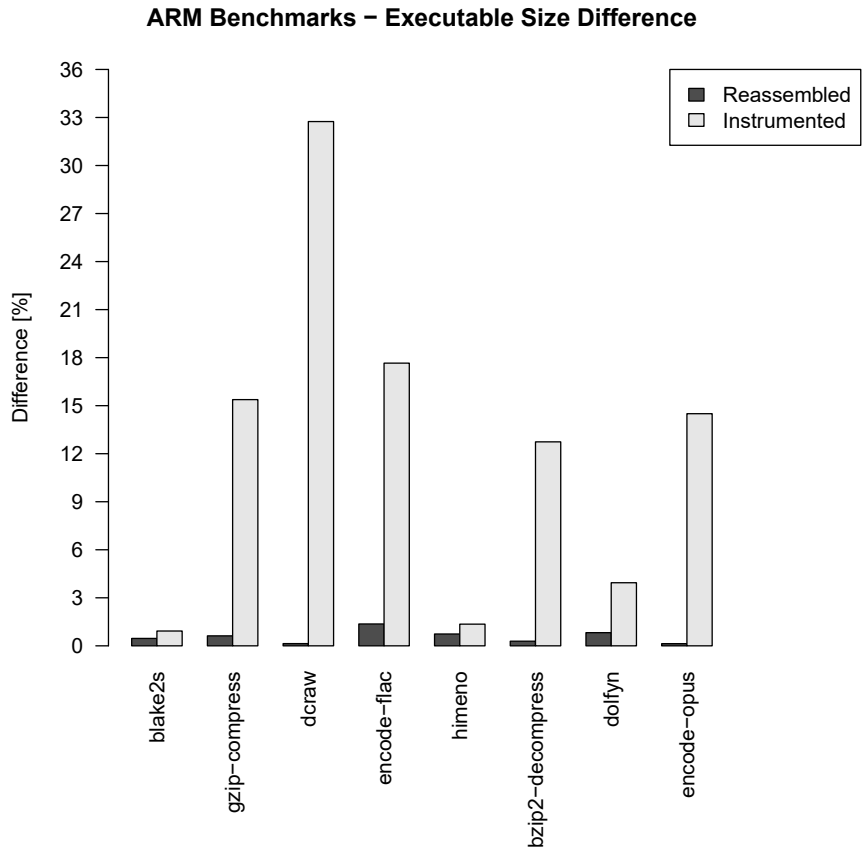

Figure 4.9: Benchmark executable file size difference for ARM binaries

4.3 Closed source application

The main advantage of being able to directly apply code modifications to executable files is the possibility of instrumenting programs where the original source code is not publicly available. As a proof of such concept, the instrumentation process was also performed on the proprietary *satisfiability modulo theories* (SMT) solver *SatEEn* [44], which is only distributed as a 32 bit x86 ELF binary.

The performance measurements for the program were obtained by running it against a subset of 62 compatible (the solver only supports input files employing up to version 1.2 of the SMT definition language while currently available benchmarks are written in the newer 2.x standard; compatibility was dictated by the possibility of performing such backwards translation) tests selected from the `mathsat` category

of the QF_LIA SMT benchmarking suite. These tests showed a 0.52% overhead caused just by the reassembling procedure and a 11.61% overhead for the instrumented version of the application, similar values to the ones obtained with the other benchmarking utilities.

It should be anyway noted that each of the 62 tests requires a separate invocation of the solver, which means that it must be factored in such performance penalty value the fact that the initial key generation routine had to be executed multiple times and, consequently, the effective instrumentation overhead is slightly lower (the running time for a single SMT problem is in the order of milliseconds thus making not possible to have precise measurements without performing a batch execution). On the subject of executable file size, the reassembled version presents a size decrease of 3.17% (for the same reasons explained in the previous section) while an increase of 15.18% is recorded for the instrumented version.

4.4 Attack simulation

This section describes a simple simulation (on x86 64 bit architecture) of how the presented instrumentation counteracts a buffer overflow attack aimed at modifying the program's behavior by means of code chaining.

The proposed example (sample code in Appendix A.4) is a small program in which the attack target is represented by a global flag `state` which is set to 1 by default. Such flag can be accessed by the function `set_state` which uses the value passed in the EDI register to modify it. The code also contains another function which uses the completely insecure `gets` routine to read a character string from the standard input. The attack's objective is the exploit the unchecked input pipe in order to change the value of the `state` flag to 0.

Fundamental in order to successfully mount the described attack is to locate suitable *gadgets* to set desired register values. In this instance, such action is performed by the instruction block "`cmp EAX, 0x488DFF31; mov EBX, EAX`" (this was voluntarily added to the code in order to facilitate this demonstration but similar blocks are not unlikely to be found in real executables) which, if interpreted in a unaligned manner with a byte of offset, is translated as "`xor EDI, EDI; lea ECX, [RAX - 0x77]; ret`", effectively providing a *gadget* to

set the value of EDI to 0.

The attack can then be performed by passing the correctly formatted input to the standard input of the program. The string is composed by: a certain number of non-significant characters used to overflow the `gets` buffer and reach the first return address, the address of the *gadget* zeroing EDI, the address of the `set_state` function to store the value of EDI to the flag and, finally, the original return address of the routine containing `gets` to proceed with the rest of the execution. By doing so, the program's execution follows the "address chain" and modifies `state` as desired.

Considering now an instrumented version of the executable, the previously described violation is no longer effective. The attack fails at the first return statement encountered after the buffer overflow: the return address decryption mechanism in place at such exit point tries to decrypt an already valid address resulting in corrupted data which causes a memory access violation (segmentation fault) when employed by the `ret` instruction.

Supposing an attacker is however able to circumvent this first line of defense, the *gadget* employed above is no more usable since it has been transformed using an *alignment sled* as shown in Listing 4.1.

```
cmp EAX, 0x488DFF31
jmp .+11
nop
...
nop
mov EBX, EAX
```

Listing 4.1: Gadget code block with sled

```
xor EDI, EDI
lea ECX, [RAX - 0X15]
or  [RAX - 0X6F6F6F70], EDX
nop
...
nop
mov EBX, EAX
```

Listing 4.2: Gadget misaligned code block with sled

As a consequence, also the misaligned interpretation of the block (Listing 4.2) changes and the execution is forced back to the correct

alignment and is going to be stopped at the next address decryption before the successive return statement (or, in this particular case, a memory violation would already happen when executing the indirect addressing of the `or` instruction).

Chapter 5

Discussion

5.1 Limitations

The most significant limitation of the proposed approach is introduced by the heuristic nature of the recompilation/instrumentation framework. The reconstruction of symbolic information necessitates the formulation of assumptions on the meaning of the constant values located in the code and data sections of executables and non-functioning programs are easily obtained when these does not hold. Such situations were encountered multiple times (especially for values in the `.rodata` and `.data` sections) while performing tests on benchmarking binaries, which required a significant time investment in debugging and manual analysis in order to gather more information on the structure of targeted executables and use it in the symbol recognition process to correct errors.

Such procedure may also suffer from code containing unusual manually crafted constructions or code obfuscations. An example could be the process of loading an address into a register split among multiple instructions: the operation "`mov RAX, 0x400400`" could also be written as "`mov RAX, 0x200000; shl RAX, 1; add RAX, 0x400`" and, while the constant in the first case would be recognized as an address and substituted with a label, the latter case would not match any recognition pattern hence resulting in a corrupted memory pointer if any relocation happens in the recompilation process.

Moreover, further complications arise when the targeted binaries are produced from high level languages and/or special compilers employing more sections of the executable to store meta-information. For

example, executables generated starting from the C++ language contain specific sectors used for the management of object constructors and destructors and the propagation of exceptions.

Some limitations also apply to the effectiveness of the employed instrumentation techniques. The procedure does not eliminates *free branches* from the instrumented binaries but prepends to these protection blocks. This means that an attacker could manage to jump over such checks and still be able to exploit the short *gadgets* still present in the binary, although the range and capabilities of the practicable attacks would be very limited.

5.2 Future work

One of the purposes of the described project has been to test the feasibility of extending instrumentation frameworks and techniques over different system architectures. Given the encouraging results, the adaptation to other different architectural types can be investigated and implemented in order to produce a more complete and generic platform.

Moreover, on the subject of the problems in symbol reconstruction pointed out in the previous section, an interesting approach which could be used to tackle the issue is the employment of machine learning algorithms. The current implementation relies on strict patterns which identify constant values as addresses just by performing checks on whether these are contained in the virtual address space of the analyzed binary. If a more sophisticated classifier could be trained in order take into account more information other than such checks (e.g. the position of the pointer in the file, its distance to other pointers, etc.), it could produce more a flexible evaluation and possibly a lower number of false positives, hence improving the robustness of the recompilation procedure.

Another improvement could be achieved by implementing the *frame cookie* protection mechanism (Section 3.2.3) to be located on the program's stack, as in the original *G-Free* design [2]. The choice of using a separate thread-local stack was dictated by the complexity and criticality of the analysis of stack references which would have been required to compensate the layout change in the program's stack caused by the insertion of the cookie. Such analysis must be very accurate, es-

pecially for optimized executables, and small mistakes can easily lead to non-functioning applications. As a consequence, the proposed implementation trades some of the protection capabilities of the defense technique with an improved stability and reliability of the instrumentation process.

Furthermore, to take into consideration other possible applications, the assembler code recovery engine employed as the foundation to apply the described instrumentation technique is suitable for different utilizations. An example could be the implementation of a layout randomizer to modify the functions' positions in the code section for executable binaries not compiled with native support for ASLR [29]. Another case, especially applicable for architectures like ARM where data values can be found embedded among instructions in the `.text` section, is the production of executables with encrypted code targeted to processors supporting live decryption in the instruction fetching stage [45]. Yet another possible use of the binary rewriting capabilities of the tool is the implementation of *Software Fault Isolation* (SFI) models, where code modification are required to guard potentially dangerous operations.

5.3 Ethics

When it comes to research ethics, the subject of *reverse engineering* (RE) represents a sort of gray area. Disassembling and modifying a software product on the development of which an individual or a company invested time and/or capital may be sometimes deemed as a violation of intellectual property. In the case of the proposed project, the RE work is aimed to the improvement of the targeted application by providing protection against the exploitment of possible design flaws. In this regard, such activity should be considered ethical and source of innovative value.

5.4 Conclusion

This thesis project has discussed the design and implementation of a static binary rewriting tool capable of instrumenting executables compiled in a Linux environment for x86 and ARM processors against *Return Oriented Programming* attacks. The results obtained from the

developed prototype tool show that such process is in fact feasible and the overhead introduced on targeted applications by such procedure is on average acceptably low. Although strict testing and source code analysis remain the best practices for the development of secure applications, the described methodology can be a valid alternative when such proceedings are not applicable. Furthermore, the flexibility of the employed instrumentation framework can make it applicable to more software analysis and security fields other than ROP defense.

Source code of the described tool is publicly available at the *GitHub* repository `https://github.com/piax93/uroboros` on the `allpy` branch.

Bibliography

[1] Marco Prandini and Marco Ramilli. "Return-Oriented Programming". In: *IEEE Security Privacy* 10.6 (November 2012), pp. 84–87. ISSN: 1540-7993. DOI: 10.1109/MSP.2012.152.

[2] Kaan Onarlioglu et al. "G-Free: Defeating Return-oriented Programming Through Gadget-less Binaries". In: *Proceedings of the 26th Annual Computer Security Applications Conference*. ACSAC '10. Austin, Texas, USA: ACM, 2010, pp. 49–58. ISBN: 978-1-4503-0133-6. DOI: 10.1145/1920261.1920269. URL: http://doi.acm.org/10.1145/1920261.1920269.

[3] S. N. Bokhari. "The Linux operating system". In: *Computer* 28.8 (August 1995), pp. 74–79. ISSN: 0018-9162. DOI: 10.1109/2.402081.

[4] Hongjiu Lu. *Elf: From the programmer's perspective*. Tech. rep. 500 Westchester Avenue White Plains, NY 10604, USA: NYNEX Science and Technology, May 1995.

[5] Michael Matz et al. "System V Application Binary Interface". In: *AMD64 Architecture Processor Supplement, Draft v0* 99 (2013).

[6] ARM Limited. *ARM Information Center: Conditional execution*. URL: http://infocenter.arm.com/help/index.jsp?topic=%2Fcom.arm.doc.dui0068b%2FChdehgih.html (visited on 10/08/2017).

[7] Yves Younan. *25 Years of Vulnerabilities: 1988–2012*. Tech. rep. Sourcefire Vulnerability Research Team, 2013.

[8] Solar Designer. *Getting around non-executable stack (and fix)*. Tech. rep. 1997.

[9] Hovav Shacham. "The Geometry of Innocent Flesh on the Bone: Return-into-libc Without Function Calls (on the x86)". In: *Proceedings of the 14th ACM Conference on Computer and Communications Security*. CCS '07. Alexandria, Virginia, USA: ACM, 2007, pp. 552–561. ISBN: 978-1-59593-703-2. DOI: 10.1145/1315245.1315313. URL: http://doi.acm.org/10.1145/1315245.1315313.

[10] Erik Buchanan et al. "When Good Instructions Go Bad: Generalizing Return-oriented Programming to RISC". In: *Proceedings of the 15th ACM Conference on Computer and Communications Security*. CCS '08. Alexandria, Virginia, USA: ACM, 2008, pp. 27–38. ISBN: 978-1-59593-810-7. DOI: 10.1145/1455770.1455776. URL: http://doi.acm.org/10.1145/1455770.1455776.

[11] Tim Kornau. "Return oriented programming for the ARM architecture". PhD thesis. Master's thesis, Ruhr-Universität Bochum, 2010.

[12] Derek Bruening and Saman Amarasinghe. "Efficient, transparent, and comprehensive runtime code manipulation". PhD thesis. Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[13] Bryan Buck and Jeffrey K Hollingsworth. "An API for runtime code patching". In: *The International Journal of High Performance Computing Applications* 14.4 (2000), pp. 317–329.

[14] Chi-Keung Luk et al. "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation". In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: ACM, 2005, pp. 190–200. ISBN: 1-59593-056-6. DOI: 10.1145/1065010.1065034. URL: http://doi.acm.org/10.1145/1065010.1065034.

[15] Nicholas Nethercote and Julian Seward. "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation". In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '07. San Diego, California, USA: ACM, 2007, pp. 89–100. ISBN: 978-1-59593-633-2. DOI: 10.1145/1250734.1250746. URL: http://doi.acm.org/10.1145/1250734.1250746.

[16] Andrew R. Bernat and Barton P. Miller. "Anywhere, Any-time Binary Instrumentation". In: *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*. PASTE '11. Szeged, Hungary: ACM, 2011, pp. 9–16. ISBN: 978-1-4503-0849-6. DOI: 10.1145/2024569.2024572. URL: http://doi.acm.org/10.1145/2024569.2024572.

[17] M. A. Laurenzano et al. "PEBIL: Efficient static binary instrumentation for Linux". In: *2010 IEEE International Symposium on Performance Analysis of Systems Software (ISPASS)*. March 2010, pp. 175–183. DOI: 10.1109/ISPASS.2010.5452024.

[18] Kapil Anand et al. "Decompilation to compiler high IR in a binary rewriter". In: *University of Maryland, Tech. Rep* (2010).

[19] Shuai Wang, Pei Wang, and Dinghao Wu. "Reassembleable Disassembling". In: *24th USENIX Security Symposium (USENIX Security 15)*. Washington, D.C.: USENIX Association, 2015, pp. 627–642. ISBN: 978-1-931971-232. URL: https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/wang-shuai.

[20] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. "Dynamic Integrity Measurement and Attestation: Towards Defense Against Return-oriented Programming Attacks". In: *Proceedings of the 2009 ACM Workshop on Scalable Trusted Computing*. STC '09. Chicago, Illinois, USA: ACM, 2009, pp. 49–54. ISBN: 978-1-60558-788-2. DOI: 10.1145/1655108.1655117. URL: http://doi.acm.org/10.1145/1655108.1655117.

[21] Ping Chen et al. "DROP: Detecting Return-Oriented Programming Malicious Code". In: *Information Systems Security: 5th International Conference, ICISS 2009 Kolkata, India, December 14-18, 2009 Proceedings*. Ed. by Atul Prakash and Indranil Sen Gupta. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 163–177. ISBN: 978-3-642-10772-6. DOI: 10.1007/978-3-642-10772-6_13. URL: https://doi.org/10.1007/978-3-642-10772-6_13.

[22] Stephen Checkoway et al. "Return-oriented Programming Without Returns". In: *Proceedings of the 17th ACM Conference on Computer and Communications Security*. CCS '10. Chicago, Illinois, USA: ACM, 2010, pp. 559–572. ISBN: 978-1-4503-0245-6. DOI:

10.1145/1866307.1866370. URL: http://doi.acm.org/10.1145/1866307.1866370.

[23] Martín Abadi et al. "Control-flow Integrity". In: *Proceedings of the 12th ACM Conference on Computer and Communications Security*. CCS '05. Alexandria, VA, USA: ACM, 2005, pp. 340–353. ISBN: 1-59593-226-7. DOI: 10.1145/1102120.1102165. URL: http://doi.acm.org/10.1145/1102120.1102165.

[24] Chao Zhang et al. "Practical Control Flow Integrity and Randomization for Binary Executables". In: *Proceedings of the 2013 IEEE Symposium on Security and Privacy*. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 559–573. ISBN: 978-0-7695-4977-4. DOI: 10.1109/SP.2013.44. URL: http://dx.doi.org/10.1109/SP.2013.44.

[25] Tigist Abera et al. "C-FLAT: Control-FLow ATtestation for Embedded Systems Software". In: *CoRR* abs/1605.07763 (2016). URL: http://arxiv.org/abs/1605.07763.

[26] Ghada Dessouky et al. "LO-FAT: Low-Overhead Control Flow ATtestation in Hardware". In: *CoRR* abs/1706.03754 (2017). URL: http://arxiv.org/abs/1706.03754.

[27] Lucas Davi, Ahmad-Reza Sadeghi, and Marcel Winandy. "ROPdefender: A Detection Tool to Defend Against Return-oriented Programming Attacks". In: *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ASIACCS '11. Hong Kong, China: ACM, 2011, pp. 40–51. ISBN: 978-1-4503-0564-8. DOI: 10.1145/1966913.1966920. URL: http://doi.acm.org/10.1145/1966913.1966920.

[28] Crispin Cowan et al. "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-overflow Attacks". In: *Proceedings of the 7th Conference on USENIX Security Symposium - Volume 7*. SSYM'98. San Antonio, Texas: USENIX Association, 1998, pp. 5–5. URL: http://dl.acm.org/citation.cfm?id=1267549.1267554.

[29] PaX Team. *PaX address space layout randomization (ASLR)*. 2003. URL: https://pax.grsecurity.net/docs/aslr.txt (visited on 12/14/2017).

[30] 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. Washington DC, USA: ACM, 2004, pp. 298–307. ISBN: 1-58113-961-6. DOI: 10.1145/1030083.1030124. URL: http://doi.acm.org/10.1145/1030083.1030124.

[31] Nguyen Anh Quynh. "Capstone: Next-gen disassembly framework". In: *Black Hat USA* (2014).

[32] Michael Larabel and Matthew Tippett. *Phoronix test suite*. 2011. URL: https://www.phoronix-test-suite.com (visited on 10/08/2017).

[33] Markku Juhani Saarinen and Jean-Philippe Aumasson. *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*. RFC 7693. Internet Engineering Task Force, November 2015. URL: https://tools.ietf.org/html/rfc7693.

[34] Antaeus Feldspar. *An explanation of the deflate algorithm*. 2011. URL: http://www.gzip.org/deflate.html (visited on 10/08/2017).

[35] Peter Deutsch. *DEFLATE Compressed Data Format Specification version 1.3*. RFC 1951. Internet Engineering Task Force, May 1996. URL: https://tools.ietf.org/html/rfc1951.

[36] M. Burrows and D. J. Wheeler. *A block-sorting lossless data compression algorithm*. Tech. rep. Digital Equipment Corporation, 1994.

[37] Jean-Marc Valin, Koen Vos, and Timothy Terriberry. *Definition of the Opus Audio Codec*. RFC 6716. Internet Engineering Task Force, September 2012. URL: https://tools.ietf.org/html/rfc6716.

[38] *HP Pavilion g6-2338sl Notebook PC Product Specifications*. URL: https://support.hp.com/hk-en/document/c03723312 (visited on 11/02/2017).

[39] *SanDisk Ultra II SSD*. URL: https://www.sandisk.com/home/ssd/ultra-ii-ssd (visited on 11/02/2017).

[40] Marvell. *MV78460 ARMADA® XP Highly Integrated Multi-Core ARMv7 Based System-on-Chip Processors*. URL: https://www.marvell.com/docs/embedded-processors/assets/marvell-embedded-processors-armada-xp-mv78460-hardware-specifications-2014-07.pdf (visited on 11/02/2017).

[41] Linux Kernel Organization. *perf: Linux profiling with performance counters*. URL: https://perf.wiki.kernel.org (visited on 11/06/2017).

[42] Micro:bit Educational Foundation. *micro:bit*. URL: http://microbit.org (visited on 11/07/2017).

[43] ARM Limited. *Cortex-M0*. URL: https://developer.arm.com/products/processors/cortex-m/cortex-m0 (visited on 11/07/2017).

[44] Hyondeuk Kim et al. *SatEEn: SMT solver*. URL: http://vlsi.colorado.edu/~hhkim/sateen (visited on 11/16/2017).

[45] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. "Countering code-injection attacks with instruction-set randomization". In: *Proceedings of the 10th ACM conference on Computer and communications security*. ACM. 2003, pp. 272–280.

Appendix A

Code snippets

A.1 Examples

All the following examples were compiled using the *GNU Compiler Collection* (GCC) version 5.4 on a x86 64 bit system with all optimizations and optional features disabled. Behaviour and characteristics (e.g. stack allocation) of the resulting binaries may vary when created with different compilers.

A.1.1 Stack overflow

This code reads a character string from the standard input and then prints the value of another string variable. The input is read without checking buffer boundaries and, consequently, the stack can be corrupted using long enough input values.

```
#include <stdio.h>

int main(){
    char string1[6];
    char string2[6] = "hello";
    scanf("%s", string1);
    puts(string2);
    return 0;
}
```

Listing A.1: Program vulnerable to stack overflow

The memory addresses of the two string variables are 16 bytes away from each other so after the insertion of 16 characters it is possible to edit the contents of `string1`.

Normal execution:

```
$> ./a.out
world
hello
```

Stack overwriting:

```
$> ./a.out
12345678123456 78world
world
```

A.1.2 Return Address Poisoning

Using the same methodology shown in the previous section, this is a simple example of how the control flow of a program can be hijacked by overwriting return addresses. The code has two functions `normal()`, which contains a vulnerable buffer operation, and `secret()`, which is supposed to never be called during a normal execution.

```
#include <stdio.h>

void secret(){
    printf("All security is now disabled\n");
}

void normal(){
    char buffer[6];
    printf("Tell me something...\n");
    scanf("%s", buffer);
}

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

Listing A.2: Program vulnerable to control flow hijacking

When `normal()` is invoked the stack contains: 16 bytes allocated for local variables, 8 bytes for the pushed base pointer and another 8

byte for the return address pointing to the middle of the main function. In order to redirect the execution to the `secret()` function, it is necessary to input 24 bytes followed by the little endian representation of the virtual address of that routine.

Normal execution:

```
$> ./a.out
Tell me something...
Hello
```

Overwriting return address:

```
$> echo -ne "123456781234567812345678\x86\x05\x40
   " | ./a.out
Tell me something...
All security is now disabled
```

A.2 Key generation

Below is the C code of the simple key generation procedure which is used at the beginning of program execution in order to create the keys then employed in return address encryption and indirect branch protection. The function simply reads two word-size unsigned integers from the Linux pseudo-random generator device and saves these in global memory. The instrumentation process selects the assembler translation of this procedure for the architecture at issue and prepends it to the `main` function of the targeted application.

```
#include <fcntl.h>
#include <unistd.h>

static unsigned long xorkey;
static unsigned long cookiekey;

void keygen() {
```

```
        size_t len = sizeof(long);
        int fd = open("/dev/urandom", O_RDONLY);
        if( fd < 0
         || read(fd, &xorkey, len) != len
         || read(fd, &cookiekey, len) != len ) {
            fail("Failed to generate keys");
        }
        close(fd);
    }
```

Listing A.3: C code of key generation routine

A.3 Frame cookie

Below are the assembler implementations of the frame cookie technique employed to guard indirect branches.

A.3.1 x86 64 bit

```
func:                           ; function entry
  ...                           ; return address encryption
  push RAX                      ; save RAX
  push RBX                      ; save RBX
  add  fs:[cstack@tpof], 1      ; increment stack size
  mov  RBX, fs:[cstack@tpof]    ; mov stack size to RBX
  mov  RAX, [cookiekey]         ; load key
  xor  RAX, $funcID             ; generate cookie and
  xor  RAX, RBX                 ; save it on cookie stack
  mov  fs:[cstack@tpof + 8 * RBX], RAX
  pop  RBX                      ; restore RBX
  pop  RAX                      ; restore RAX

  ...
  push RAX                      ; save RAX
  mov  RAX, fs:[cstack@tpof]    ; load stack size
  xor  RAX, fs:[cstack@tpof + 8 * RAX]
  xor  RAX, $funcID             ; extract key from cookie
  cmp  RAX, [cookiekey]         ; compare with correct key
  pop  RAX                      ; restore RAX
  jne  fail                     ; jump to fail if not ok
  call *RBX                     ; indirect branch
```

```
  ...
  sub   fs:[cstack@tpof], 1     ; decrement stack size
  ...                           ; return address decryption
  ret                           ; return
```

Listing A.4: x86 64 bit frame cookie

A.3.2 x86 32 bit

```
func:                           ; function entry
  ...                           ; return address encryption
  push EAX                      ; save EAX
  push EBX                      ; save EBX
  add   gs:[cstack@ntpof], 1    ; increment stack size
  mov   EBX, gs:[cstack@ntpof]  ; mov stack size to EBX
  mov   EAX, [cookiekey]        ; load key
  xor   EAX, $funcID            ; generate cookie and
  xor   EAX, EBX                ; save it on cookie stack
  mov   gs:[cstack@tpof + 8 * EBX], EAX
  pop   EBX                     ; restore EBX
  pop   EAX                     ; restore EAX

  ...
  push EAX                      ; save EAX
  mov   EAX, gs:[cstack@ntpof]  ; load stack size
  xor   EAX, gs:[cstack@ntpof + 8 * EAX]
  xor   EAX, $funcID            ; extract key from cookie
  cmp   EAX, [cookiekey]        ; compare with correct key
  pop   EAX                     ; restore EAX
  jne   fail                    ; jump to fail if not ok
  call *EBX                     ; indirect branch

  ...
  sub   gs:[cstack@ntpof], 1    ; decrement stack size
  ...                           ; return address decryption
  ret                           ; return
```

Listing A.5: x86 32 bit frame cookie

A.3.3 ARM

```
func:                           ; function entry
  ...                           ; return address encryption
  push {r0, r1, r2}             ; save r0, r1, r2
```

```
mrc  p15, 0, r2, c13, c0, 3  ; load TLS address
movw r1, #:lower16:cstack(tpoff)
movt r1, #:upper16:cstack(tpoff)
add  r2, r1
movw r0, #:lower16:cookiekey ; load key
movt r0, #:upper16:cookiekey
ldr  r0, [r0]
movw r1, #:lower16:funcID    ; load function ID
movt r1, #:upper16:funcID
eor  r0, r1                  ; generate cookie
ldr  r1, [r2]                ; load stack size
add  r1, #1                  ; increment stack size
str  r1, [r2]                ; store stack size
eor  r0, r1                  ; combine with cookie
str  r0, [r2, r1, lsl #2]    ; store cookie
pop  {r0, r1, r2}            ; restore r0, r1, r2

...
push {r0, r1}                ; save r0 and r1
mrc  p15, 0, r0, c13, c0, 3  ; load TLS address
movw r1, #:lower16:cstack(tpoff)
movt r1, #:upper16:cstack(tpoff)
add  r0, r1
ldr  r1, [r0]                ; load stack size
ldr  r0, [r0, r1, lsl #2]    ; load cookie
eor  r0, r1                  ; extract key from cookie
movw r1, #:lower16:funcID
movt r1, #:upper16:funcID
eor  r0, r1
movw r1, #:lower16:cookiekey ; load correct key
movt r1, #:upper16:cookiekey
ldr  r1, [r1]
cmp  r0, r1                  ; compare value
bne  fail                    ; jump to fail if not ok
movw r0, #:lower16:.text     ; load code section addr
movt r0, #:upper16:.text
cmp  r3, r0                  ; check branch address
pop  {r0, r1}                ; restore r0 and r1
blo  .+6
orr  r3, #1                  ; force Thumb if in .text
blx  r3                      ; indirect branch

...
push {r0, r1}                ; save r0 and r1
mrc  p15, 0, r0, c13, c0, 3  ; load TLS address
movw r1, #:lower16:cstack(tpoff)
movt r1, #:upper16:cstack(tpoff)
```

```
add  r0, r1
ldr  r1, [r0]                ; load stack size
sub  r1, #1                  ; decrement size
str  r1, [r0]                ; store stack size
pop  {r0, r1}                ; restore r0 and r1
...                          ; return address decryption
bx   lr                      ; return
```

Listing A.6: ARM stack cookie

A.4 Attack simulation

Simple program employed in the demonstration in Chapter 4 of how instrumentation prevents gadget chaining attacks.

```
#include <stdio.h>

int state = 1;  // 1 = locked, 0 = unlocked

void set_state(int val){
    state = val;
}

int somerandomcode(){
    int a = 5, b = 7;
    int c = a * b;
    printf("Hello: %d\n", c);
    __asm__ volatile(
      "cmp $0x488dff31, \%eax\n"
      "mov \%eax, \%ebx\n"
    );
    return 2 * c;
}

void verybad(){
    // gets does not perform any check and
    // reads input until newline of EOF
    char buffer[6];
    gets(buffer);
```

```
}

int main(){
    int a = 5, b = 10;
    verybad();
    printf("State: %d\n", state);
    return 0;
}
```

Listing A.7: C code of the program used for attack demonstration

Appendix B

Large tables

B.1 Benchmark Results

The following tables list all the numeric results of the performed benchmarks. Every test was run multiple times (10 or more) and the three best times were recorded in order to produce as fair scores as possible.

B.1.1 x86 64 bit executables

Program	Execution Time [s]		
	Original	**Reassembled**	**Instrumented**
blake2s	0.541	0.545	0.551
	0.544	0.546	0.551
	0.540	0.543	0.553
gzip-compress	14.308	14.245	15.836
	14.173	14.241	15.868
	14.246	14.277	15.859
dcraw	5.242	5.318	5.894
	5.253	5.296	5.848
	5.243	5.404	5.852
encode-flac	3.024	3.073	3.348
	3.040	3.066	3.344
	3.043	3.059	3.352
himeno	11.542	11.581	11.603
	11.522	11.585	11.589
	11.558	11.604	11.613

bzip2-decompress	13.895 13.612 13.684	13.744 13.968 13.874	14.577 14.709 14.627
dolfyn	0.523 0.526 0.527	0.525 0.531 0.527	0.641 0.637 0.645
encode-opus	7.507 7.548 7.551	7.632 7.552 7.527	8.436 8.377 8.468

B.1.2 x86 32 bit executables

Program	**Execution Time [s]**		
	Original	**Reassembled**	**Instrumented**
blake2s	0.544 0.541 0.543	0.546 0.545 0.548	0.559 0.554 0.548
gzip-compress	14.026 14.001 14.040	14.143 14.085 14.125	14.189 14.193 14.200
dcraw	5.632 5.591 5.609	5.670 5.663 5.671	7.134 7.190 7.130
encode-flac	3.565 3.597 3.600	3.592 3.609 3.675	3.734 3.729 3.762
himeno	11.029 11.005 11.044	11.076 11.114 11.138	11.160 11.112 11.118
bzip2-decompress	16.420 16.311 16.342	16.392 16.482 16.296	16.899 16.931 16.809
dolfyn	0.731 0.737 0.739	0.737 0.733 0.741	0.788 0.789 0.794

encode-opus	10.464 10.520 10.511	10.566 10.479 10.486	11.659 11.603 11.650
SatEEn	0.960 0.974 0.968	0.977 0.957 0.983	1.081 1.069 1.089

B.1.3 ARM executables

Program	**Execution Time [s]**		
	Original	**Reassembled**	**Instrumented**
blake2s	4.537 4.526 4.536	4.549 4.550 4.548	4.636 4.639 4.631
gzip-compress	73.559 73.430 73.486	74.156 74.283 74.209	76.947 76.822 77.006
dcraw	44.709 44.781 44.732	45.124 45.143 45.206	48.567 48.518 48.548
encode-flac	91.063 90.934 90.945	91.097 91.105 91.099	92.193 92.206 92.186
himeno	252.438 252.253 251.224	252.563 252.638 252.278	252.158 254.055 252.058
bzip2-decompress	62.396 62.416 62.417	62.515 62.505 62.509	62.179 62.210 62.196
dolfyn	6.920 6.926 6.920	6.921 6.920 6.629	6.890 6.900 6.896
encode-opus	63.607 63.611 63.591	63.654 63.617 63.633	63.633 63.735 63.761

B.2 File sizes

The following tables list the file sizes of the employed benchmark. All binaries were stripped of all symbol information.

B.2.1 x86 64 bit executables

Program	Filesize [bytes]		
	Original	Reassembled	Instrumented
blake2s	14552	14568	14688
gzip-compress	118672	118688	155624
dcraw	416888	412808	584928
encode-flac	534752	526480	637160
himeno	14576	14600	14704
bzip2-decompress	87600	87600	99976
dolfyn	852208	848112	1216872
encode-opus	295312	291216	336376

B.2.2 x86 32 bit executables

Program	Filesize [bytes]		
	Original	Reassembled	Instrumented
blake2s	9708	13944	14008
gzip-compress	117416	118684	135116
dcraw	440408	434160	520232
encode-flac	572952	565396	671948
himeno	9720	9844	14000
bzip2-decompress	90808	91412	99656
dolfyn	925016	918484	1164308
encode-opus	290168	288832	338040
SatEEn	513708	497424	591688

B.2.3 ARM executables

Program	Filesize [bytes]		
	Original	**Reassembled**	**Instrumented**
blake2s	13836	13900	13964
gzip-compress	55844	56192	64432
dcraw	276536	276920	367088
encode-flac	327188	331656	384960
himeno	9748	9820	9880
bzip2-decompress	66216	66408	74652
dolfyn	527560	531900	548352
encode-opus	200016	200288	229016