

Compile-Time Control Flow Integrity as a Defence Against Return-Oriented Programming

SUBMITTED 2018

JIM FIELDING

CANDIDATE NO. 149106

DEGREE: COMPUTER SCIENCE BSC

SUPERVISOR: DR MARTIN BERGER

Declaration

This report is submitted as part requirement for the degree of Computer Science at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged.

Signature:

Jim Fielding

Acknowledgements

The author wishes to thank Dr. Martin Berger for his encouragement throughout the project, his advice and support whilst supervising the project and for introducing me to the world of Scala.

Table of Contents

Introduction	6
1.1 x86 Assembly Language	7
Compilers	9
2.1 The Analysis Phase	11
2.2 The Code Generation Phase	12
2.3 Control Flow.....	14
2.3.1 Labels	14
2.3.2 Branches	15
Return-Orientated Programming	16
3.1.1 System Libraries	16
3.1.2 Gadgets	17
3.1.3 ROP Chains.....	20
Control-Flow Integrity.....	21
4.1 Control-Flow Graphs	21
4.2 Maintaining the Integrity of Control-Flow	24
Performance Counters (Perf Counters).....	27
5.1 Perf.....	27
5.2 Last-Branch Register.....	28
5.3 Existing Solutions	29
The Project	32
6.1 Project Scope	32
6.1.1 The Compiler	32
6.1.2 Return-Orientated Programming Style Attack.....	32
6.1.3 Defence Against a ROP Style Attack	33
6.2 Project Aims.....	34
6.3 Approach	34
6.4 Report Structure.....	35
Professional Considerations.....	36
7.1 Public Interest	36
7.2 Professional Competence and Integrity	36
7.3 Duty to Relevant Authority	37
7.4 Duty to Profession.....	37
7.5 Ethical Considerations	37
Build.....	38
8.1 Compiler	38
8.1.1 Lexer.....	38
8.1.2 Parser	40
8.1.3 Intermediate-Code Generation	42
8.1.4 Code-Generation	43
8.1.5 Testing.....	43
8.2 Control-Flow Graph Generation	44
8.3 Control-Flow Labels.....	47

Testing and Data Analysis	48
7.1 Compile-Time Cost	48
7.2 Runtime Cost.....	51
7.2.1 kBouncer Comparison	55
7.2.2 ROPecker Comparison.....	55
7.3 Attacks Foiled.....	57
7.4 Viability	57
Evaluation	58
8.1 The Implementation	58
8.2 Cost Evaluation	58
Conclusion.....	59
Bibliography	60
Appendix A.....	63

Chapter 1

Introduction

Cyber security is an arms race between attackers and defenders. Red team vs. Blue team, black-hat vs white-hat the opposing sides are constantly to exploit or mitigate the others latest advances. The landscape of cyber security is a rapidly growing one, with the dependence integrated system for many of our world's cities and utilities to the wealth of customer data stored on server space there have never been more targets for the budding hacker to exploit. Advances in CPU design and manufacturing leave us pushing the boundaries of Moore's law [1] but leave in their path dense sets of legacy instruction sets that remain to ensure compatibility with modern systems that provide a means of exploitation.

This arms race is not dissimilar to a game of chess, the attacker holding the white pieces and the defender the black. The attacker merely needs to find one gap, one mistake to exploit in order to win the game or in this case, exploit the system, hijack the session or steal the data. Conversely, the defenders must plan for all modes of attack and stay one step ahead of the attacker, reacting to any new strategies they may encounter. In this, the defender perhaps has the harder job, instead of looking for one hole they fight to plug many. With this in mind, the broader the coverage of a defence mechanism in the cyber security world the better.

In the past, attackers installed malicious software on victims' computers and then ran it to do harm. In recent years defenders have changed modern operating systems from Windows to iOS so attackers can no longer install software for execution. This stopped most traditional forms of cyber-attacks. In response attackers invented Return-Oriented Programming (ROP) [2], an ingenious attack that does not install new software: instead existing data (e.g. images or music files) on the attacked machine is reinterpreted as attack software. All existing defences were unable to prevent ROP. Fortunately, ROP has a weakness: using existing files as attack tools leads to programs that jump in memory more than normal software. Control-Flow Integrity (CFI) [3] detects this unusual jumping and can be used to stop attacks using ROP. CFI is continuously advancing with Intel as recently as June 2017 [4] releasing a preview of their proposal for Control-flow Enforcement Technology (CET) an advancement on standard CFI.

Compilers, ROP and CFI are integral to this report and to mirror this have dedicated chapters in due course. To provide greater context to several of the figures in this report and much of the discussion, it is important first to consider x86 assembly.

1.1 x86 Assembly Language

The x86 assembly language, first introduced in 1978 with the 16-bit 8086 microprocessor [5], is the language used by the majority of modern CPUs that find their way into laptops and desktops. It was the inception of Intel and AMD both of whom have been competing in this sector. With such popularity, the instruction sets of each evolution of processor have been backdated to ensure support for legacy systems. Although this clearly has its benefits, an increasingly dense instruction set means that a given binary could be interpreted as a number of different instructions, clearly an exploitable quality.

There are a number of different variations for the syntax of x86 with the most popular being Intel [6] and AT&T [7]. Throughout this report there are examples of both. *Figure 3* depicts an abstraction of the memory available within the architecture, with a high variety of different registers.

```
mov ebp, esp
sub esp, 4
mov [ebp-4], 1
mov esp, ebp
```

Figure 1 - x86 code snippet for variable assignment within a method

Figure 1 contains a small snippet of x86 code that will be used as a means to further understand the syntax of the language. The above code is using the Intel syntax. The majority of instructions have two operands that are separated using a comma, these operands can be registers, literals or indirect address references (a reference to the address pointed to by the contents of `[_]`). An arithmetic instruction such as `sub` will store the result of performing the subtraction into the accumulator. This is the `EAX` register. The `EBP` and `ESP` registers are the stack pointer and base pointers and are responsible for ensuring correct control flow between stack frames. With the variations in syntax these can be written as shown above or using AT&T syntax would be written as `%EAX`. A `mov` instructions will both one operand into the other. In Intel syntax the ordering of operands is `dest, source`. In AT&T this is reversed. The above program allocates space for a local variable and stores the value 1 in it before returning. Its behaviour is easier to visualise in a higher-level language as shown in *Figure 2*.

```
def main() = {
    x := 1
}
```

Figure 2 - Higher-level example of Figure 1

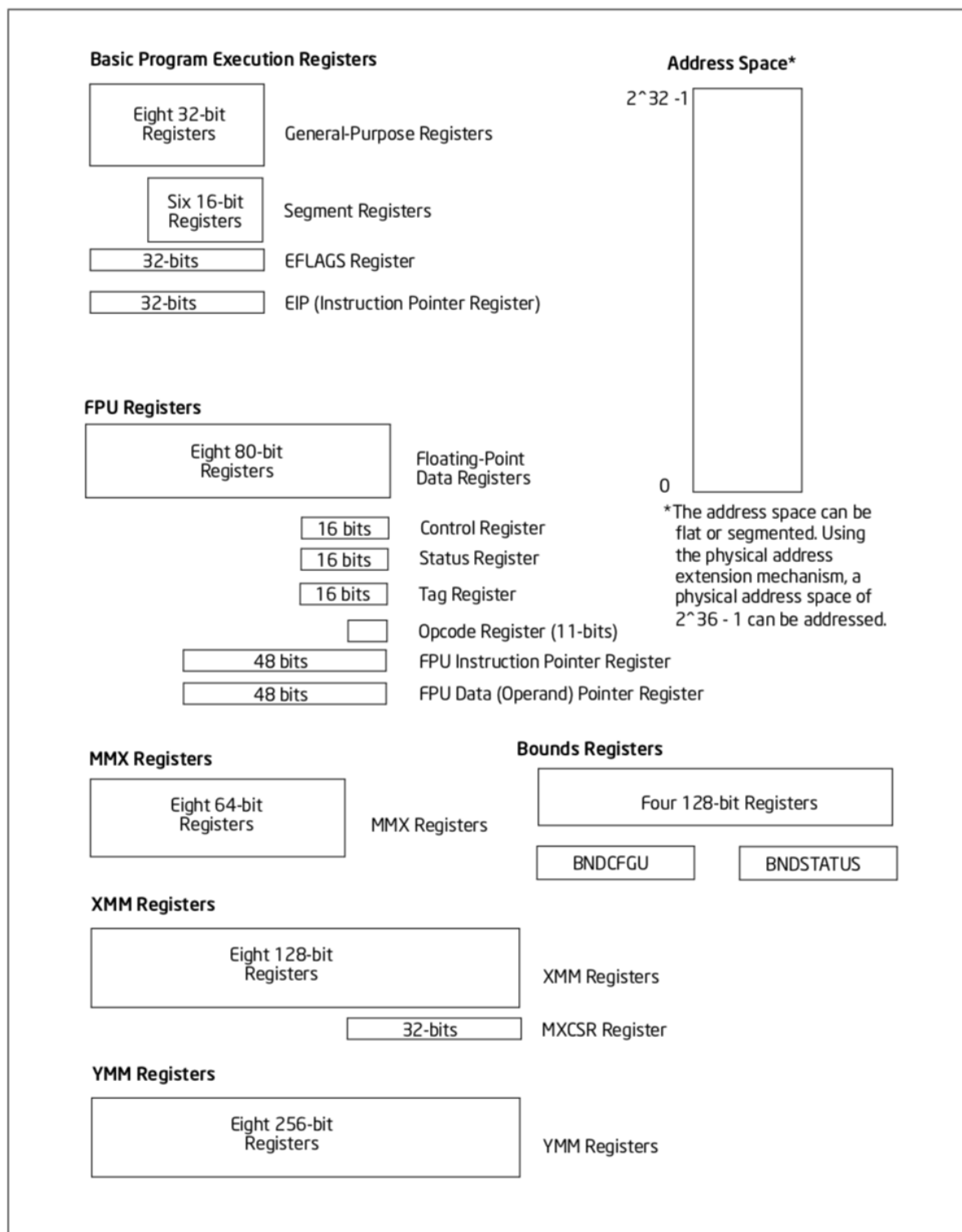


Figure 3 - Memory layout of x86 CPU [8]

Compilers

A compiler functions to translate one programming language into another as demonstrated in *Figure 4*. Although not exclusively, a compiler often translates a high-level language (e.g. Java [9], Swift [10], Haskell [11], Scala [12]) into a low-level language (e.g. ARM [13], x86 [8]). A high-level language is designed to be human friendly whereas a low-level language machine friendly. A compiler could therefore be viewed as bridging the abstraction gap between machine convenient low-level languages and the human convenient high-level languages.

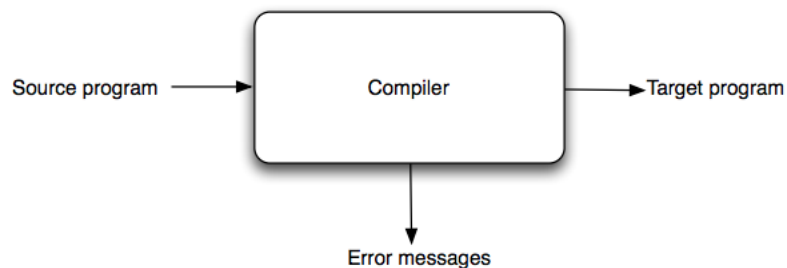


Figure 4 - Simple Compiler [14]

The abstraction gap encapsulated the syntactic but more importantly the semantic gap between the source and target language. Although the semantics in the given language will differ the most important requirement for the compiler is to retain the semantics of the program that is being translated such that the program written in the source language and the target language behave in exactly the same manner.

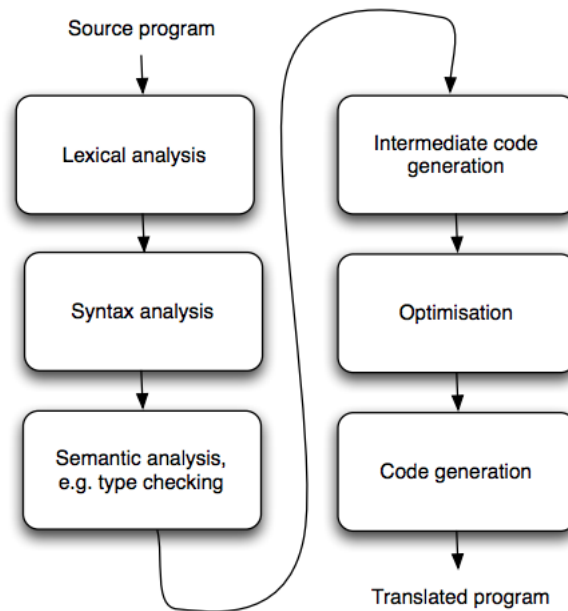


Figure 5 - Compiler Pipeline [14]

A compiler is a heavily modular model as shown by *Figure 5*. This makes building a compiler a relatively simple task, with each module performing a relatively simple task, with the exception of the optimisation phase. However, this will be considered generally outside the scope of this report. The phases involved in compilation can be grouped as in *Figure 6* to define a compiler's primary functions, code analysis and code generation.

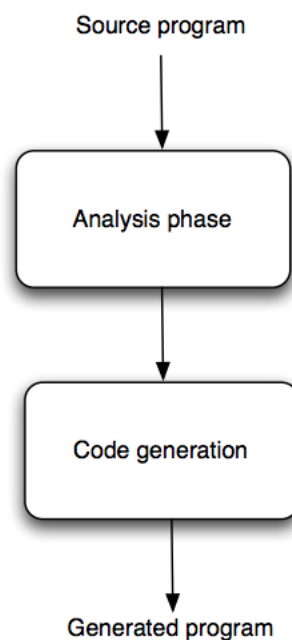


Figure 6 - Abstracted Compiler Pipeline [14]

2.1 The Analysis Phase

The code analysis phase of the compiler functions to check the correctness of the source program and to break it down into a more accessible data structure before it is translated. For a program to be deemed correct it has to be lexically, syntactically and semantically correct. The checking is done by three components:

- The lexer functions to tokenise¹ the source program into tokens, a data structure that can easily be parsed to check for syntactic correctness. In tokenising the input, the lexer ensures that the program contains only the lexemes of the source language.
- The parser functions to check that the input is syntactically correct, i.e. that it conforms to syntax rules defined by the source language. *Figure 7* demonstrates how this may be formalised using a context-free grammar. In checking syntactic correctness, the parser also ensures that all method invocation and variable references can be resolved and that all brackets are well formed. Upon parsing the tokens are translated into corresponding Abstract Syntax Trees (ASTs) a data structure that removes unnecessary syntactic information such as brackets for the rest of the compilation process.
- The type-checker ensures that the type system of the given language has not been violated, i.e. that a Boolean value is not assigned to an Integer or a String to a Double. In the context of this report this will be trivial as the source language will contain only Integers.

¹ Tokenisation splits a given input string into tokens using a boundary that delimits the end of a word (whitespace). A token defines the function of a word, in English: noun, adjective word or in a computing sense, Integer, String, If statement. This is performed as the parser only needs this information for syntactic analysis, variable and method names are not necessary in this phase.

```

PROG → DEC | DEC; PROG
DEC → def ID (VARDEC) = E
VARDEC → ε | VARDECNE
VARDECNE → ID | VARDECNE, ID
ID → (identifiers)
INT → (Integers)
E → INT
  | ID
  | if E COMP E then E else E
  | (E BINOP E)
  | (E; E)
  | while E COMP E do E
  | repeat E until E COMP E
  | ID := E
  | ID(ARGS)
ARGS → ε | ARGSNE
ARGSNE → E | ARGSNE, E
COMP → == | < | > | <= | >=
BINOP → + | - | * | /

```

Figure 7 - Context-Free Grammar for a Language

2.2 The Code Generation Phase

The code generation phase of the compiler functions to translate the source program into the target language by traversing the ASTs provided by the analysis phase. As the input program has already been proven to be well-formed this can be a relatively simple process dependent mainly on the complexities of the two languages. To maintain this relative simplicity this phase is split into three phases:

- The intermediate-code generation phase translates the ASTs into an intermediate representation (IR) that is lower-level than the source language but often not an example of genuine assembly/machine code. An IR can be represented in a number of different ways but is often done using Three-Address instructions with two example data structures shown in *Figure 8*. This translation can be performed independently of a given computer architecture and saves times when porting compilers for newer processes as it does some of the hard-lifting already.
- The optimisation phase is partly architecture specific and is the most complex phase of compilation, often containing several passes that can amalgamate to thousands of lines of code. This functions to remove unnecessary code duplication and to optimise the use of registers. This phase of compilation is outside the scope of this report, given the focus being on CFI and ROP.
- The code generation phase is wholly architecture specific and performs the final translation into the target language. It simply takes the optimised IR code and

translates this into its equivalent. Within the context of this report all code generation will produce x86 assembly code.

```

t1 = minus c
t2 = b * t1
t3 = minus c
t4 = b * t3
t5 = t2 + t4
a = t5

```

	<i>op</i>	<i>arg₁</i>	<i>arg₂</i>	<i>result</i>
0	minus	i c		t1
1	*	i b	t1	t2
2	minus	i c		t3
3	*	i b	t3	t4
4	+	t2	t4	t5
5	=	t5		a

(a) Three-address code

(b) Quadruples

Figure 8 – Alternative Data Structures for Three-Address Instruction [15]

2.3 Control Flow

Control flow is important to consider when discussing compilers. It refers to the passing of control between chunks in code memory during the execution of a program. Given that this occurs at runtime it can present some hurdles given that all possible paths of execution within a program have to be considered. The control flow of a given program can be represented using a Control-Flow Graph (CFG) [16]. The process of representing this will be addressed in the chapter on Control-Flow Integrity (CFI) given that it is integral to functioning CFI, however *Figure 9* demonstrates a skeleton example of a CFG.

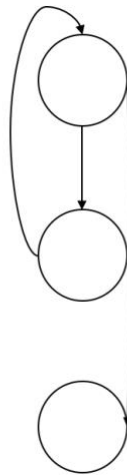


Figure 9 - Skeleton CFG

2.3.1 Labels

Alphanumeric or numerical labels can be used in a number of lower-level languages to mark the position of chunks of code in memory. Control flow in these programs is often dictated by jumps between these labels. *Figure 10* demonstrates an example code snippet with inline labelling. The nodes in *Figure 9* may represent these labels with the edges between these nodes representing the branches (passing of control flow) between them. The use of unique labels is fundamental to CFI and this will again be revisited in Chapter 4. All abundant in any given program, labels should have no side-effect on program execution but should act merely as pointers to blocks of instructions.

```

        jl 14
        mov [ebp-4], 2
        jmp 112
14:
15:
        mov ecx, 1
        mov edx, 2
        mov edx, eax
        cmp ecx, edx
        jg 15
        jmp 111

```

Figure 10 - x86 Snippet with in-line labels

2.3.2 Branches

A branch in a program can be defined as a jump from one position in the source code to another. Branches are always a side-effect of an executed instruction. A number of common programming constructs after translation perform a number of branching executions including:

- If-Then statements
- If-Then-Else statements
- While statements
- Repeat Until statements
- Method invocations

Although the syntax of the instructions may vary from language to language, instructions that alter control flow can be classified into two categories:

- Direct branches
- Indirect branches

A direct branch refers to a movement in the source code that will move to the same destination each time it is executed regardless of the given input. In other words, the destination for this branch is always known. This can be achieved by leveraging the before-mentioned labels that are inserted into the source program.

An indirect branch refers to a movement in the source code that does not branch to a predefined location in the form of label but to an offset to a memory address. Instructions causing an indirect branch could not know the location they are branching to until runtime and therefore can be considered to be 'unaware' of their destination. Although not exclusively, these are often found at the end of a function call as it looks to return to the function from which it was called. It is these indirect branches that ROP looks to leverage to gain control over the flow of a given program.

Return-Orientated Programming

Within the current cyber security climate ROP can be considered to be one of the more troubling issues within the last ten years and has therefore triggered a considerable response from a number of the big players in the technology industry from both a hardware and software perspective. Hardware defences have come to the forefront first with the introduction of Data Execution Prevention (DEP) [17] and the NX-bit [18]. In 2015 Microsoft introduced Control Flow Guard (CFG) into Visual Studio 2015 [19] and more recently Intel have released a proposal for their answer to this, Control-Flow Enforcement Technology (CET) [4]. Both of these software implementations have something in common with the topic of this report in that they focus on locking down control-flow.

ROP exploits a system by creating Turing complete programs from code already in memory. With jumps between enough small snippets of code, a malicious program can be executed by an attacker. The code snippets known as gadgets are often but not exclusively found in the libc library. This method for attacking a system was originally proposed by Sebastian Krahmer in his paper *“x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique”* [20] referring to the technique and ‘borrowed code chunks’. He concluded that return-into-libc style attacks would no longer work in the face of current mitigation techniques. However, this was disproved by Hovav Shacham in his paper *“The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)”* [2], coining the name Return-Orientated Programming. As noted by Krahmer and Shacham the x86 architecture is vulnerable to this type of attack, therefore any reference to ROP henceforth will be in the context of an attack on the x86 architecture, specifically Intel x86 processors.

Despite the ever-evolving battle between ROP exploits and ROP mitigation the overall goal of ROP has remained the same, the attacker looks to build a stack of legitimate code locations ending in a return that in turn, following execution, will grant the attacker the necessary privileges to allow them to execute their malicious payload. It is important to distinguish that ROP, in the context of this report, is considered a tool to allow the execution of malicious code and not as a means of performing the entire malicious execution. Although not impossible, this is complex and worthy of its own report. These legitimate code locations are often found within the system libraries on a given system.

3.1.1 System Libraries

As referenced in both Krahmer’s and Shacham’s papers the C standard library, often referred to as libc [21], is dense and can be very useful if it can be leveraged. More importantly, given that the C programming language is abundant in the code base of most operating systems, this library is widely available. libc covers a range of topics in what it can be used for including:

- Data Types
- Character classification
- String

- Mathematics
- File input/output
- Date/Time
- Localisation
- Memory allocation
- Process control
- Signals
- Alternative tokens

Although Krahmer showed that the functions in this library can no longer be leveraged against current defences, Schacham showed that its diverse set of responsibilities meant that it contained an abundance of code chunks (employing Krahmer's 'borrowed code' philosophy), that if chained together, can grant an attacker control. These code chunks are referred to as Schacham as gadgets.

3.1.2 Gadgets

A gadget is simply a pointer to a sequence of machine/assembly level instructions that already exist within memory. Any sequence of instructions could be considered a gadget and by themselves do not necessarily perform a useful role. However, when combined together to create a denser sequence of instructions, hardware-based defences such as DEP and the NX-bit can be leveraged to give an attacker write privileges.

Naturally, there are plenty of instruction sequences that bare little use in the context of ROP. A useful gadget functions to specify a value that is to be placed onto the stack that will make use of an instruction sequence in libc or another location. They often perform xor, load or jump operations. As noted by Schacham the majority of useful gadgets end with an instruction that pops a value from the stack and then returns. Consider below three simple examples given by Schacham.

3.1.2.1 Load

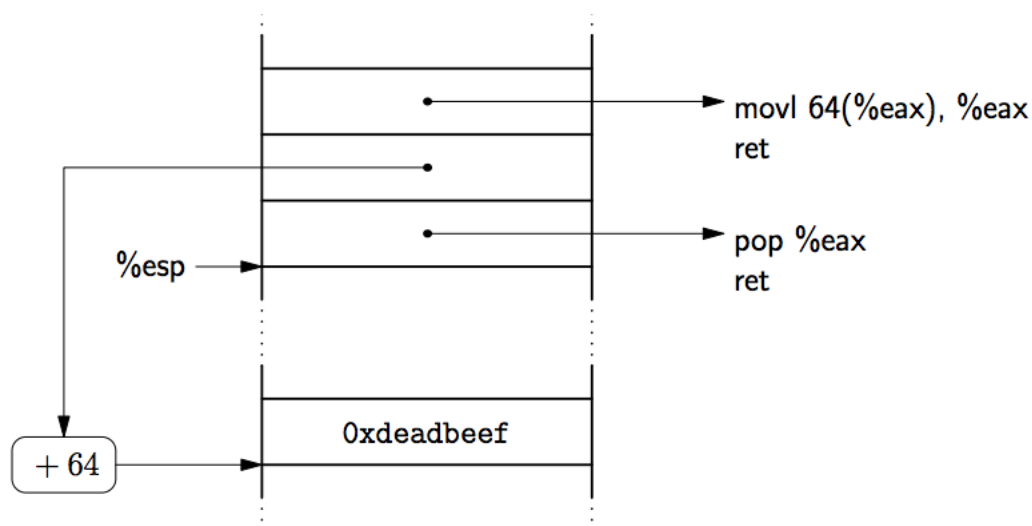


Figure 11 - Load ROP Gadget [2]

The gadget shown in *Figure 11* shows how a load instruction can be leveraged alongside a pop and ret instruction to point execution to the attacker's next gadget. As 0xdeadbeef is popped off of the stack, %esp (the stack pointer) is advanced past the end of the gadget causing the ret instruction to point to the next gadget placed above it, continuing the attack's execution.

3.1.2.2 Add

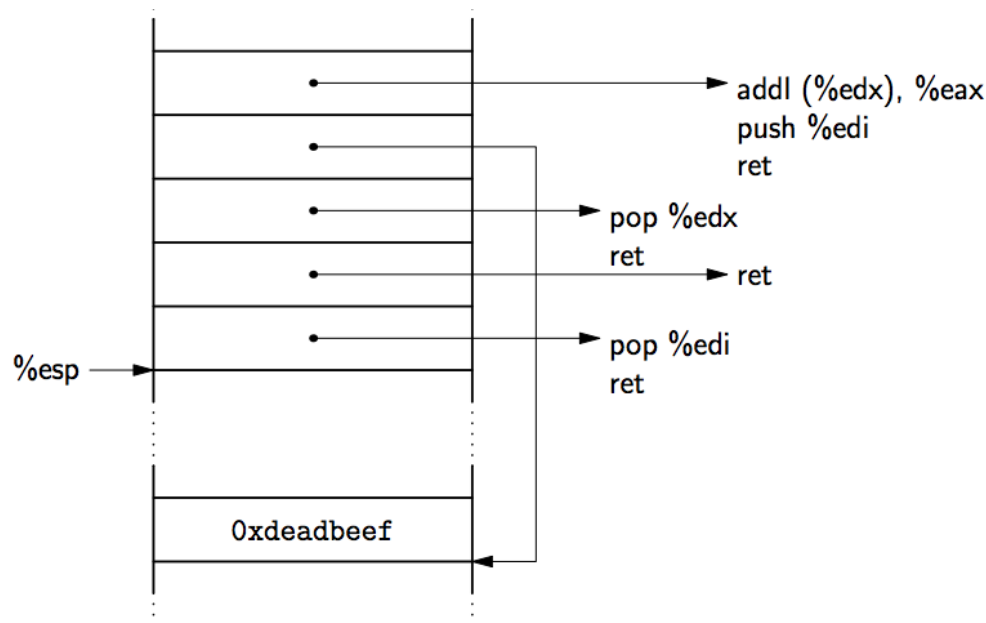


Figure 12 - Add ROP Gadget [2]

The gadget shown in *Figure 12* shows how an add can be leveraged to continue execution onto the attacker's next gadget. This gadget demonstrates how a ret instruction can be used much like a no-operation instruction (nop), functioning only to advance the stack pointer (%esp). Once the add operation is completed, this will again return into the next gadget situated below this gadget. *Figure 13* demonstrates the final behaviour of the gadget after execution.

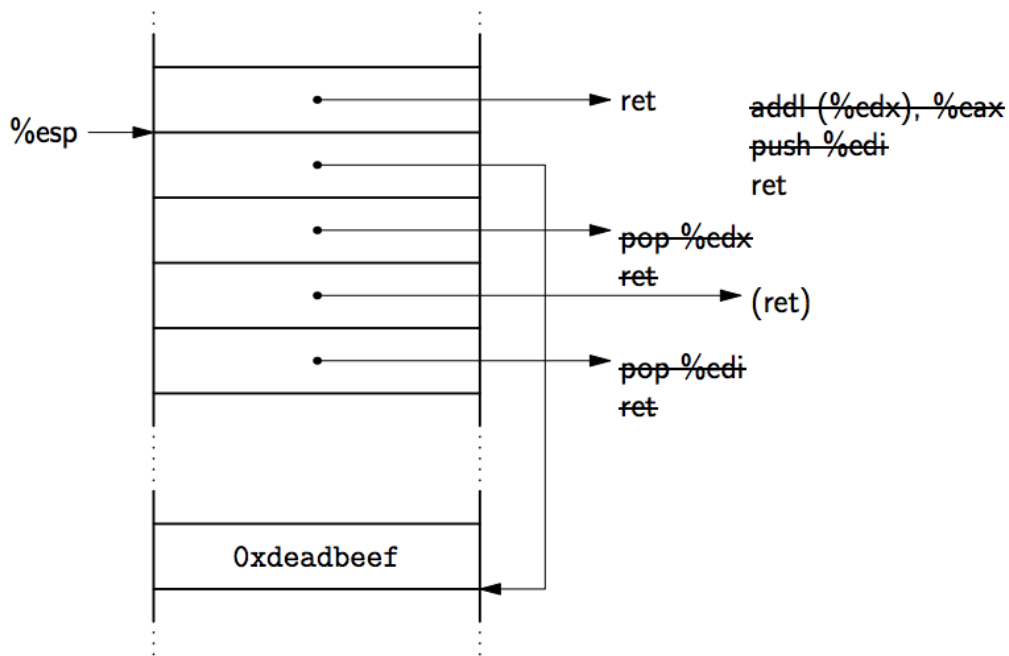


Figure 13 - Add Gadget Semantics [2]

3.1.2.3 Jumps

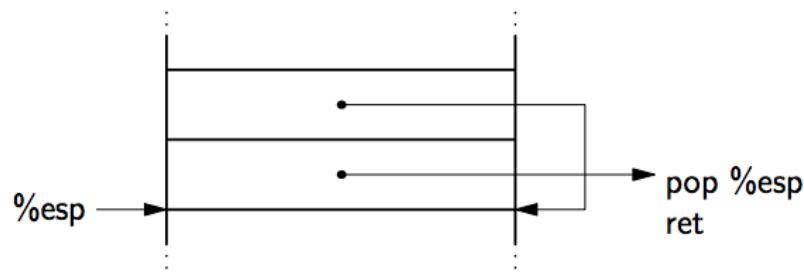


Figure 14 - Jump ROP Gadget [2]

The gadget shown in *Figure 5* is a simple gadget that demonstrates how the pop and ret instructions can be leveraged to produce an unconditional jump to the next gadget within the attacker's sequence. To leverage this the value of %esp has to be changed to point to the location of the new gadget. This can also be used to create infinite loops which can aid an attacker gain the privileges they seek by causing an overflow. Figure _ demonstrates this infinite loop. These types of gadgets although simple are very important to an attacker as they allow easy flow between each of the attacker's gadgets. This full sequence of gadgets that return into each other is often referred to as a ROP chain.

3.1.3 ROP Chains

A ROP chain encompasses a sequence of gadgets that when executed in sequence, given the attacker the power to grant themselves write privileges and therefore execute their malicious payload. The construction of an effective ROP chain is outside the scope of this project, however *Figure 15* demonstrates what an effective ROP chain may look like, allowing an attacker to execute their arbitrary shell code.

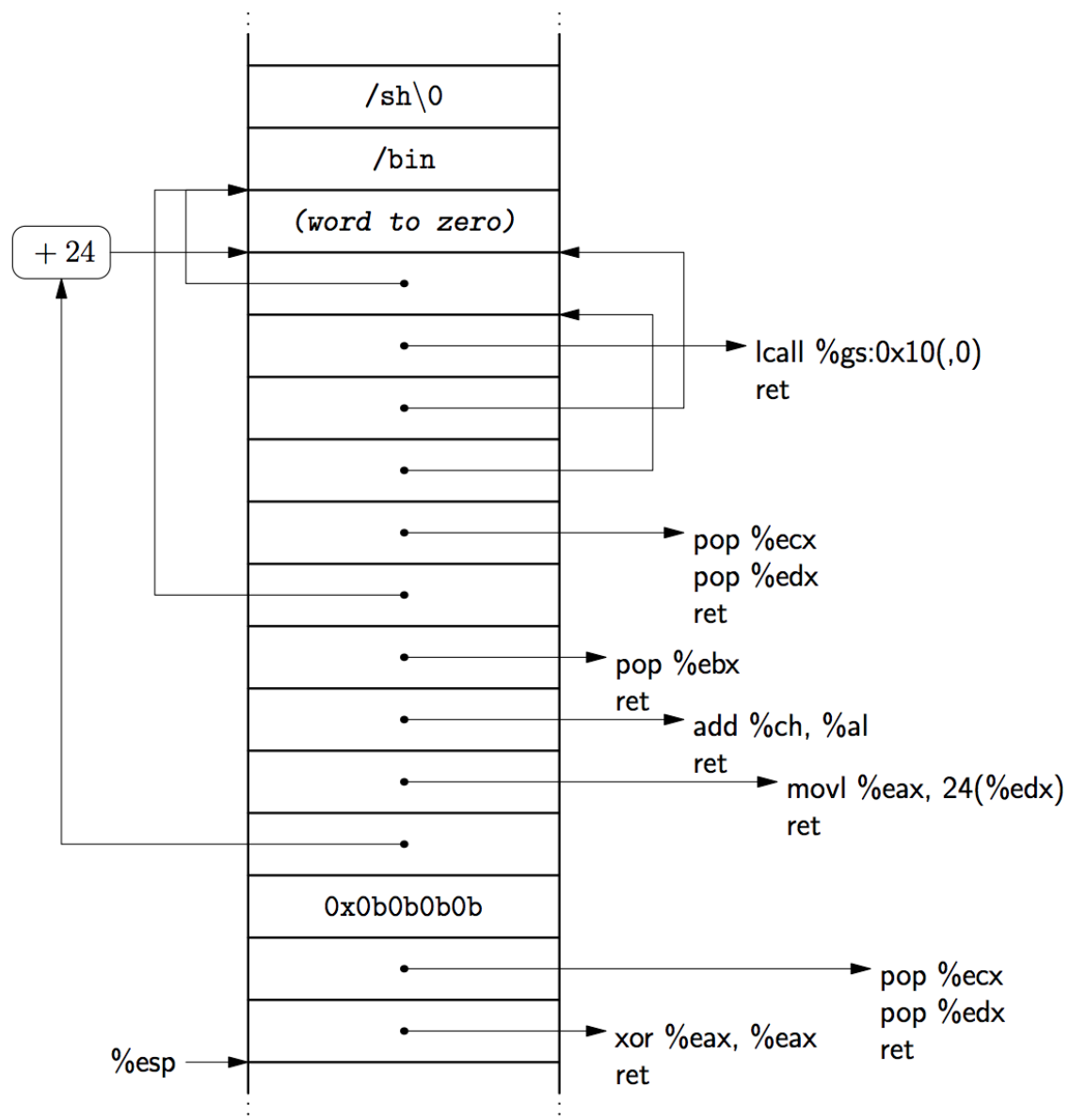


Figure 15 - Shellcode ROP Attack [2]

Control-Flow Integrity

To prevent against ROP attacks, a program can be monitored to ensure that it performs as expected, i.e. all of the jumps in memory that occur at runtime are the same as they were intended to be (at compile time if a direct functional call, or at runtime if an indirect functional call that must be calculated). In essence, the integrity of the control-flow must be maintained to ensure that a program has not been compromised. This is where the concept of Control-Flow Integrity comes in [3]. If CFI is maintained, in the majority of cases, it can be assumed that code is safe to execute as an attacker has not been able to acquire control-flow and trigger any ROP gadgets. To maintain the expected behaviour of the program's control-flow it first must be established exactly what the expected behaviour of the program was to be. This is achieved using Control-Flow graphs.

4.1 Control-Flow Graphs

A Control-Flow graph is a graph that maps the flow of control within a given program. A CFG conforms to the standard computer science definition of a graph in that it is made up of nodes and edges. Each node represents a chunk of labelled code in memory, be it a function or a block of code as part of a statement, and each edge represents a branch/jump to this section of code. All jumps in the program must move between these functions and across the provided edges. If an edge does not exist between a function and a function that it is attempting to call, say for example a return to a function in libc then the program will be halted as the control-flow has not been maintained. The Control-Flow graph for a given program can be constructed at compile time or thereafter by analysing program binaries post compilation. *Figure 16* demonstrates a code chunk and *Figure 17* its corresponding CFG.

```

1)  i = 1
2)  j = 1
3)  t1 = 10 * i
4)  t2 = t1 + j
5)  t3 = 8 * t2
6)  t4 = t3 - 88
7)  a[t4] = 0.0
8)  j = j + 1
9)  if j <= 10 goto
    i) i = i + 1
11) if i <= 10 goto
12) i = 1
13) t5 = i - 1
14) t6 = 88 * t5
15) a[t6] = 1.0
16) i = i + 1
17) if i <= 10 goto

```

Figure 16 - IR of Three-Address Instructions [15]

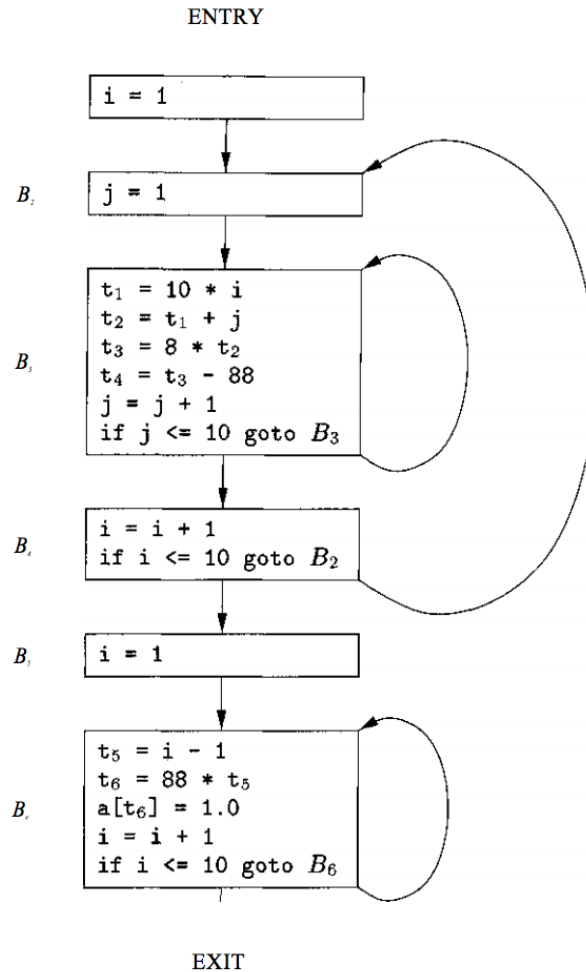


Figure 17 - Control Flow Graph of Figure 16 [15]

4.1.1 Constructing a Control-Flow Graph During Compilation

To construct a CFG the blocks of code that function as the nodes of the graph must first be identified. This construction can be done after the intermediate-code generation phase and before the code-generation phase of compilation. Code can be broken up into basic blocks by iterating over the set of three-address instructions that are produced from the intermediate-code generation. The first instruction of each basic block is referred to as a leader so a pass over the sequence of instructions should first mark each of the leaders. An instruction is a leader if it fulfils one of the following criteria [16]:

1. It is the first three-address instruction in the intermediate code.
2. It is the target of a conditional or unconditional jump.
3. It immediately follows a conditional or unconditional jump (is the 'fall through' of the jump instruction).

After this pass, each basic block can be generated by taking each leader and generating its basic block to include itself and each instruction up to and not including the next leader.

The Control-Flow Graph can then finally be constructed by determining the predecessors and successors for each given block, i.e. the blocks that pass flow of control into a block and the blocks to which a block will pass flow of control to. These can be determined by analysing the leader of each block.

A block B' is a successor of block B if:

- Block B contains a conditional or unconditional jump to block B'
- Block B' is the fall through from a conditional jump in B
- Block B' directly follows block B where B contains only a copy instruction

A block B' is the predecessor of block B if:

- The leader of block B was the target of a conditional or unconditional jump in block B'
- Block B is the fall through for a conditional jump in block B'
- Block B' contains only a copy instruction and is directly followed by block B

4.2 Maintaining the Integrity of Control-Flow

Once the control-flow graph has been constructed, checks must be enforced to make sure that each change in control-flow within the program at runtime conforms to this graph. At this juncture, it is important to consider three assumptions that must hold to ensure that control-flow integrity is a successful defence mechanism [3]:

- Uniqueness of IDs: the ID's used for nodes in the graph (functions) must remain unique in that they are only used for the function of ID-checks and IDs and are not used anywhere else in code memory.
- Non-executable data: data should be exactly that and should not be capable of being executed as if it were code and not data. The majority of modern architectures support this using something similar to Microsoft's Data Execution Prevention (DEP).
- Immutable code: Once compiled code should be immutable, i.e. it should not be capable of being rewritten at runtime to alter the semantics (and control-flow) of the program. The majority of modern architectures implement this using Executable Space Protection (ESP), an example of this is the hardware implementation of the NX (Non-executable) bit.

These checks can be implemented using the rewriting of code to introduce inline-reference monitors (IRMs) into the code base. These IRMs are simply labels (labelled as DATA in the code as to not be executable) that represent the nodes in the CFG. The ID label is placed at the address before the function code and any call to this function pointer will jump to this address and a check will be made. An example of this implemented can be seen in *Figure 18*

Bytes (opcodes)	x86 assembly code	Comment
8B 44 24 04 ...	mov eax, [esp+4]	; first instruction ; of destination code
can be instrumented as (a):		
78 56 34 12 8B 44 24 04 ...	DD 12345678h mov eax, [esp+4]	; label ID, as data ; destination instruction
or, alternatively, instrumented as (b):		
3E 0F 18 05 78 56 34 12 8B 44 24 04 ...	prefetchnta [12345678h] mov eax, [esp+4]	; label ID, as code ; destination instruction

Figure 18 - IRM Labelling [3]

ID labels must be provided for calls and returns as a function will not be aware of its caller and to maintain the integrity of control-flow a check must also be implemented here to ensure that the function is returning to the address of the function that originally called it and a jump-to-libc that an attacker has introduced. *Figure 19* and *Figure 20* below shows an example implementation of a check for a function pointer call and a function return.

Bytes (opcodes)	x86 assembly code	Comment
FF 53 08	call [ebx+8]	; call a function pointer
is instrumented using prefetchnta destination IDs, to become:		
8B 43 08 3E 81 78 04 78 56 34 12 75 13 FF D0 3E 0F 18 05 DD CC BB AA	mov eax, [ebx+8] cmp [eax+4], 12345678h jne error_label call eax prefetchnta [AABBCCDDh]	; load pointer into register ; compare opcodes at destination ; if not ID value, then fail ; call function pointer ; label ID, used upon the return

Figure 19 - Label Check on a Function Call [3]

Bytes (opcodes)	x86 assembly code	Comment
C2 10 00	ret 10h	; return, and pop 16 extra bytes
is instrumented using prefetchnta destination IDs, to become:		
8B 0C 24 83 C4 14 3E 81 79 04 DD CC BB AA 75 13 FF E1	mov ecx, [esp] add esp, 14h cmp [ecx+4], AABBCCDDh jne error_label jmp ecx	; load address into register ; pop 20 bytes off the stack ; compare opcodes at destination ; if not ID value, then fail ; jump to return address

Figure 20 - Label Check on a Return to Function Call [3]

Abadi, Budiu, Erlingsson and Ligatti concluded in their original CFI paper [3] that the overheads caused by its implementation were 'tolerable'. This paper will look to further investigate this and compare it with an alternative anti-ROP method known as perf counters.

Performance Counters (Perf Counters)

An alternative method in defending against ROP is the use of performance counters and more specifically the Last Branch Recording mechanism that Intel has introduced into its newer ranges of processors [22]. Last Branch Recording (LBR) allows for the tracing of indirect branches and utilises two characteristics of a typical ROP attack:

- The main goal of a ROP attack is to eventually perform a system call that will allow the attacker to execute their malicious code
- The majority of ROP instructions end in a return, i.e. an indirect branch in memory

The Last Branch Recording is restricted to the number of previous branches that it can store [23]. Performing a check every time a ROP gadget calls a return would produce an intolerable overhead. Instead, as a string of ROP gadgets is eventually attempting to make a system call an LBR check is only required when there is a context switch, i.e. an attempt to jump to and execute a system call. This LBR check can then recover the indirect branches that led to the system call and compare these to see if there were any abnormal control transfers. If an abnormal transfer is found, the program can be halted, and the system call avoided.

5.1 Perf

Perf is a profiler tool for Linux based system (2.6+) [24] that offers a rich set of commands that allow a user to collect and analyse performance and trace data that is stored by a CPU's performance counters. Perf runs off events and the hardware events that it recognises can be seen in *Figure 21*. Given its ability to monitor these events at runtime, Perf can be very useful in determining and comparing the runtime 'cost' of executing a particular program. Number of cycles, number of instructions, volume of branches can all help indicate if a program's runtime cost is acceptable.

<code>cpu-cycles</code> OR <code>cycles</code>	[Hardware event]
<code>instructions</code>	[Hardware event]
<code>cache-references</code>	[Hardware event]
<code>cache-misses</code>	[Hardware event]
<code>branch-instructions</code> OR <code>branches</code>	[Hardware event]
<code>branch-misses</code>	[Hardware event]
<code>bus-cycles</code>	[Hardware event]

Figure 21 - Perf Hardware Events [24]

Moreover, Perf can be used to access (given the correct permissions have been set) a number of hardware registers that can further enrich this analysis. The Last-Branch Register (LBR) is an example of this can be leveraged as a defence against ROP.

5.2 Last-Branch Register

The LBR or an equivalent has seen its introduction on the processors of the leading competitors in the CPU business since 2012. Intel specifically introduced the Last-Branch Register and have increased the number of LBRs as the processor generations advance as can be seen in *Figure 22*.

CPU Generation	No. of Last-Branch Registers
Ivy Bridge	4
Haswell, Broadwell	8
Skylake	16
Kaby Lake	32

Figure 22 - LBRs in Intel CPUs

Utilising Perf, the LBRs can be sampled [25] and then filtered to only record certain branches. As can be seen in *Figure 23* the LBR could be filtered to show only a function call or system call, which could be useful when identifying ROP-like behaviour given that most ROP chains look to end in a system call.

Name	Meaning
any_call	any function call or system call
any_ret	any function return or system call return
ind_call	any indirect branch
u	only when the branch target is at the user level
k	only when the branch target is in the kernel
hv	only when the target is at the hypervisor level
in_tx	only when the target is in a hardware transaction
no_tx	only when the target is not in a hardware transaction
abort_tx	only when the target is a hardware transaction abort
cond	conditional branches

Figure 23 - Perf Branch Filtering [24]

Moreover, if the list of branches produced can be backdated in some fashion, a program could look at control flow and in a similar fashion to CFI, check to see if the control-flow of the original program has been violated. The contents of the LBRs at a given time during execution would be expected to reflect *Figure 24*.

The use of the LBR does hold one caveat, the addressed stored in each LBR can only be accessed in Ring 0 (Kernel mode) and therefore requires the context switch and the relevant permissions to be able to do so. This report, in an attempt to provide a reasonable means for comparison with CFI, investigate if the LBR could be leveraged somehow in a similar fashion to CFI to ensure control-flow is maintained.

	Branch	Target
00	7380274A	738027D7
01	7C348C05	7C348D16
...
15	7623A6FC 5	7623A6CD

Figure 24 - LBR Contents

5.3 Existing Solutions

A number of existing solutions already demonstrate how the LBRs can be utilised as a defence against ROP by identifying ROP like behaviour. Both of the given solutions are implemented as kernel modules and run alongside the execution of a program as well as performing some analysis of each program's binaries pre-execution. These existing solutions are kBouncer [23] and ROPecker [26].

5.3.1 kBouncer

kBouncer utilises the ability to filter stored branches and upon a transfer of control from user to kernel space (a context switch) will check the last N indirect branches to decide if the system call is part of a ROP exploit or benign as demonstrated in *Figure 25*.

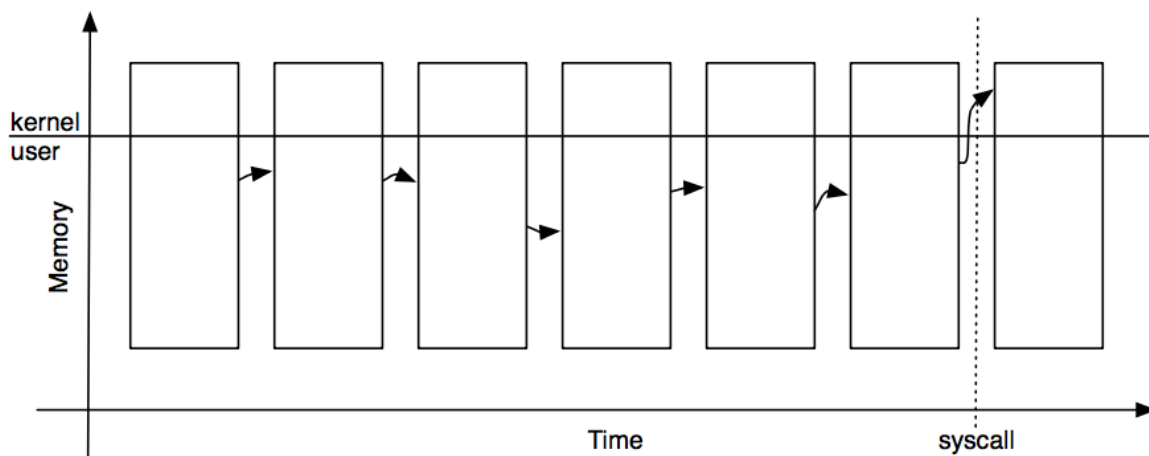


Figure 25 - Transfer of Control Flow From User to Kernel Space [23]

As demonstrated in *Figure 26* kBouncer runs as a kernel module so will be running concurrently with the executing program. Whenever a call to a WinExec is encountered, the context switch creates a detour and first checks the LBR and will only call the WinExec if the call is seen to be benign and not ROP-like.

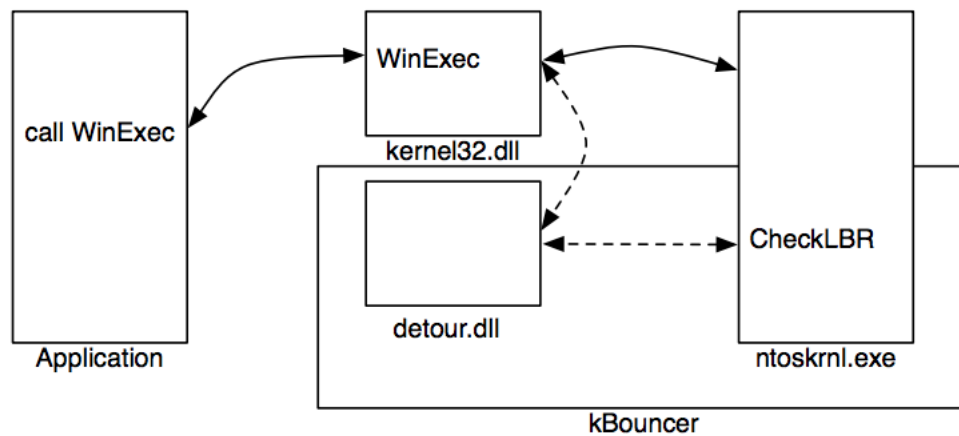


Figure 26 - kBouncer Architecture [23]

5.3.2 ROPecker

ROPecker, unlike kBouncer functions using two separate phases. The first phase that ROPecker goes through is a pre-processing phase. This process extracts all instruction information from the compiled program (offsets, alignment, types) as well as all of the shared libraries that it depends on. All potential gadgets are also identified and their potential impact on the stack and CPU instruction pointers are accessed. All of this information is stored in a database. The pre-processing of shared libraries such an interaction with libc only has to be performed once and the information can then be reused.

The second phase is during runtime with ROPecker using a sliding window of a set size. All code pages outside of this sliding window are set to be non-executable. Whenever a call is made to code outside of this window the logic shown in *Figure 27* is followed to determine if the program is ROP-like. If it is found to be benign then the sliding window moves to the new location, if ROP is detected then program execution is halted.

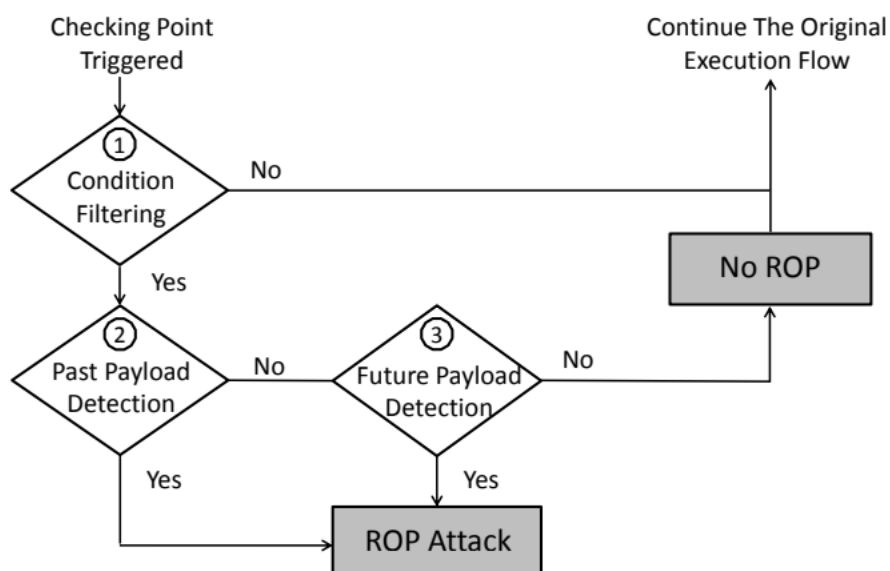


Figure 27 - ROPecker ROP Detection Logic [26]

In this model, ROPecker utilises the LBR in the Past Payload Detection phase, checking to see if previous branches are both indirect and pointing to a pre-identified gadget in the pre-processing database.

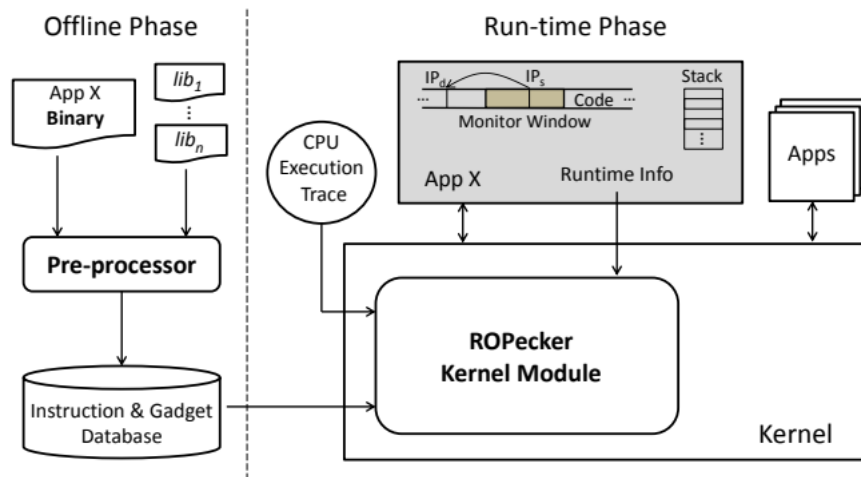


Figure 28 - ROPecker Architecture [26]

ROPecker is also implemented as a kernel module with the pre-processing phase performed offline, completely separate to program execution as depicted in *Figure 28*. Then in a similar fashion to kBouncer the detection phase using the sliding window in kernel mode constantly monitoring the branches in the executing program.

The Project

6.1 Project Scope

Before laying out the aims of this project it is important to define the scope of the project itself given the vast nature of the subject area. The project itself spans three main subject areas:

1. The compiler
2. A Return-Oriented Programming style attack
3. Defence against a ROP style attack

6.1.1 The Compiler

An effective compiler works not only to translate a source language into a target language without a loss or change in semantics but also to optimise this process to produce the most efficient set of low-level instructions as a result. The optimisation phase of a compiler can contain many passes and often is the amalgamation of thousands and thousands of lines of code. Given that this project looks to serve as an investigation into Control-Flow Integrity and not as a proposal for a new programming language and/or compiler, the optimisation phase of the compilation process will not be considered and therefore omitted. The desired translation from source to destination language can still be achieved without this, given that whole optimal the destination code is not a concern within the scope of this project.

6.1.2 Return-Oriented Programming Style Attack

A ROP style attack contains a number of different phases. An attacker cannot first inject their payload of gadgets into an executing program without first having a method by which to decompile a program to identify sections of instructions that can be exploited (to have their return address overwritten or to function as a gadget) or a method through which they can overwrite return addresses to jump to their gadgets. Methods for achieving these objectives and circumventing the increasingly dense set of defences to prevent this provides a wealth of research and investigation in itself. With this in mind, in the context of this project it is assumed that the attacker has the ability to decompile a given program and has the ability to alter return addresses and gain access to their identified gadgets.

In the vast inventory of cyber security defences, Control-Flow Integrity functions to detect illegitimate jumps within a program's source code and therefore already assumes that the defences that would prevent the before-mentioned luxuries given to an attacker have been circumvented. With this in mind it is safe in the evaluation of CFI to make these assumptions. Moreover, it allows a greater focus on the subject matter given the amount of work that would otherwise be required.

Moreover, an effective ROP attack cannot be performed without the production of an effective chain of gadgets. Identifying effective gadgets within a program can itself pose a reasonable challenge and this will also be considered to be outside the scope of this project. Instead, open-source tools that produce chains of gadgets from a given program will be used so that the CFI implementation can be properly tested and evaluated.

6.1.3 Defence Against a ROP Style Attack

The inventory of defence mechanisms against ROP is vast as can be seen in Figure __. To focus on the evaluation of CFI, it will be considered against methods that have a similar focus surround the flow of control within a program. These approaches could be categorised as methods that utilise the Last-Branch Register (LBR) that has found itself introduced and built upon with each iteration of modern processor. A comprehensive approach that utilises the LBR will not be implemented alongside the implementation of CFI as the existence of such projects demonstrates that the workload to produce such an implementation would be a project within itself. Although these external projects will be used as a means of comparison, some attempt will be made to provide a naïve implementation using LBR so that the relative ‘cost’ of accessing the register and then processing its contents can be weighed up against the ‘cost’ of corresponding CFI implementation.

6.2 Project Aims

There are two main aims for this project: to implement CFI into a compiler for a trivial source language that will produce x86 assembly code as the target language and to evaluate the 'cost' of implementing CFI at compile and runtime so that a conclusion can be made as to the practicality of CFI as a defense against ROP.

The first aim, developing a compiler that can implement CFI, will allow users interested in defense against ROP and cyber security as a whole to have a means of evaluating the 'cost' of CFI and to be able to analyse this data to determine if CFI is a practical method for defending against ROP or if other means of defence are more appropriate. The specific objectives for the development of this compiler are:

- Implement a standard compiler (lexing, parsing, type checking, code generation) in the programming language Scala [27]
- Implement CFI that can be toggled on or off during the code generation phase of compilation
- Implement a different defense technique in the form of perf counters to act as a comparison to the 'cost' of CFI implementation

The second aim, to evaluate the 'cost' and therefore practicality of implementing CFI as a defensive strategy will give those interested the relevant information so that they can decide to implement CFI or a different technique. The specific objectives for this analyse and evaluation are:

- Evaluate the 'cost' of implementing CFI at compile time
- Evaluate the 'cost' of implementing CFI at run time
- Consider an alternative defense mechanism in the form of perf counters (to be fully introduced in the 'Background Information' section of this report) to serve as a base for comparison. Achieved by looking at existing implementations and considering if a compiler-based implementation using this technology is achievable.
- Compare the relevant 'cost' of each technique to aid in the evaluation of the practicality of implementing CFI

6.3 Approach

To implement the compiler and the defence mechanism the programming language Scala [27] will be used. Scala is a language that attempts to combine the merits of both object-orientated and functional programming bringing with it a number of useful properties. The main reason this has been chosen is the power of pattern-matching in Scala [27]. This will be utilised throughout the compiler, utilising the combinator library [28] to use parser combinators to perform the lexing and parsing phases of compilation. Pattern-matching will be used in the code generation phases of compilation to match against AST's before code is produced. Although parser combinators in Scala are quite slow in terms of their runtime, the aim of this project is to investigate the 'cost' of implementing CFI and not to produce an efficient compiler and therefore is not of real concern in this case. Moreover, any effect on the compile time of a given program is somewhat negligible (within reason) as a program only needs to be compiled once, it is preserving a reasonable runtime that is the area of concern with this implementation.

To ensure accurate results in the investigation of the 'cost' of implementing CFI the compiler has to be exhaustively tested to ensure that it behaves as expected for all inputs. This is performed using a Scala based testing tool. The tool that will be utilised is Scala Test [29], a unit testing tool that will be used in the test-driven development of the compiler and defence mechanisms.

To dictate good software engineering practices and to somewhat automate the overall software engineering process GitHub [30] will be used to provide version control as well as easy access to the code base.

At the end of the project the quality of the work will be evaluated first by the implementation of CFI and secondly if the data provided regarding the 'cost' of implementing CFI is extensive enough to be able to make a reasonable conclusion about its practicality when considered against a perf counters approach.

6.4 Report Structure

This section outlines the proceeding structure of the rest of this dissertation.

- Professional Considerations: This section describes the ethical and professional considerations entailed in the undertaking this project and the impact if any on society.
- Build: This section outlines the build process of both the compiler and the defence mechanisms.
- Testing and Data Analysis: This section outlines the results of what compile time and runtime 'cost' the defence mechanisms have shown to demand. This analysis will be in terms of the effect on the time of compilation and code execution compared to the equivalent program without any form of ROP defence mechanism.
- Evaluation: This section will assess the quality of the results produced by the project and the project as whole.
- Conclusion: This section will conclude the project as whole and will detail the direction of any future work if necessary.

Chapter 7

Professional Considerations

The professional considerations detailed in this chapter have been adhered to throughout the entirety of this project. A conservative effort has been made to ensure that no single human participant or their data has been involved in this project.

The BCS [31] outline a Code of Conduct [32] that any individual conducting a project with the IT sector should adhere to. The following sections from the BCS Code of Conduct [32] are addressed and detailed below: Public Interest, Professional Competence and Integrity, Duty to Relevant Authority and Duty to Profession. Moreover, as this project is centred around cyber security, a topic area that will always carry ethical issues, the ethical considerations of this project are also addressed.

7.1 Public Interest

This project may at times use code libraries that have been produced by a third party. In these instances, the relevant third party or parties involved will be given proper credit and be explicitly referenced. Although this project is aimed predominantly at the computer science community and moreover those interested in ROP defence, the results may benefit other sectors in the improvement of their cyber security. In light of this, this project will adhere to Section 1(c) of the BCS Code of Conduct [32] in that the project will be conducted ‘without discrimination on the grounds of sex, sexual orientation, marital status, nationality, colour, race, ethnic origin, religion, age or disability, or of any other condition or requirement’.

7.2 Professional Competence and Integrity

Given that this project is being undertaken as part of a Computer Science BSc undergraduate degree, the project will predominantly focus on knowledge areas that are covered by the course as delivered by the University of Sussex. Any areas of knowledge that have been gained through background research and reading that are not covered by the course will be referenced to provide the relevant information for background reading. As a final year project the main goal of this project is to provide a strong learning experience for project author Jim Fielding under the supervision of Dr Martin Berger. Given the security based nature of this project, any new vulnerabilities found in the Intel x86 Architecture, Control-Flow Integrity defence mechanism [3] or Perf Counter defence mechanism [23] will be responsibly disclosed to the relevant authority. The relevant authority will be given ample time to resolve said vulnerabilities, as agreed with the relevant authority, before any information is made available to the public. In extreme circumstances where the withholding of this information may be to the detriment of society, this timescale may be exaggerated, again as agreed with the relevant authority. This will ensure that this project adheres to

Section 2(f) to “avoid injuring others, their property, reputation, or employment by false or malicious or negligent action or inaction” and Section 2(g) to “reject and will not make any offer of bribery or unethical inducement” of the BCS Code of Conduct [32]. Any criticism of this body of work are welcomed and were applicable will be included within to detail where a change has been made in accordance with this feedback.

7.3 Duty to Relevant Authority

This project will be conducted with due care and diligence in accordance to the requirements as set out by the University of Sussex. Permission has been granted to carry out this project by supervisor Dr Martin Berger and the University of Sussex having sought advice as to the relevance of the project. In conducting this project, any situation that may result in a conflict of interest between the University of Sussex will be avoided. Moreover, this project is being conducted by Jim Fielding who accepts full responsibility for all work carried out within the confines of this project. This project will not disclose or attempt to disclose any confidential information without abiding by the relevant legislation or without the permission of the University of Sussex. This project will conform to Section 3(e) of the BCS Code of Conduct [32] in that it will not “misrepresent or withhold information on the performance of products, systems or services (unless lawfully bound by a duty of confidentiality not to disclose such information), or take advantage of the lack of relevant knowledge or inexperience of others”.

7.4 Duty to Profession

The author of this project Jim Fielding will accept his duty to “uphold the reputation of the profession and not take any action which could bring the profession into disrepute” as described by Section 4(a) of the BCS Code of Conduct [32]. Moreover, he shall act with integrity and respect in his professional relationships with all members of BCS and with members of other professions with whom he works in a professional capacity. He will support fellow members in encouraging them in their further development.

7.5 Ethical Considerations

In conducting a project within the cyber security sector the ethical implications must always be considered. In the case of this project, defensive tools are being implemented and evaluated and not an attacking tool. Given the defensive nature of said tool it will be designed to have a positive impact on society and not a negative one. Moreover, given that the project focuses solely on a computer architecture (Intel x86) there are no human participants involved in the project in any capacity, the ethical consideration regarding the project’s possible effect on the individual are minimal.

Given the circumstance in which a vulnerability is discovered in either of the defence mechanisms or the Intel x86 architecture that are not yet of public knowledge, the relevant procedures will be followed as detailed in Section 2.2 of this paper to minimize the ethical impact of this discovery.

Chapter 8

Build

This section of the report focuses on how the code that supports the project was built. The code base will be discussed in three main subsections:

- The original compiler implementation
- Amended compiler implementation using CFI
- Performance counter monitoring

8.1 Compiler

The modular nature of a sequential compiler lends itself to a modular construction. Each module was created independently and then pipelined to complete the implementation.

8.1.1 Lexer

The lexer was constructed using the parser combinator library [28] that is available within Scala [12]. This effectively allows rewrite rules based on a pattern and provides a very simple construct with which to tokenise an input String. The lexer is made up of two files: `Lexer.scala` and `LexerToken.scala`. The main function of the lexer is to return a list of the `LexerToken`'s that will then be passed to the parser.

As can be seen in *Figure 29* the defining of tokens is trivial in Scala with the use of case classes and objects. Case classes and case objects are powerful as they allow for easy pattern matching on the Tokens in the parsing phase.

The tokenisation of keywords is also made trivial with the use of parser combinators as can be shown in *Figure 30*. The tokenisation of identifiers and integer literals is the only action that is not wholly trivial but is still easily achieved using the regex libraries available in Scala as depicted in *Figure 31*.

```

trait LexerToken extends Positional {

  case object T_Def extends LexerToken
  case object T_If extends LexerToken
  case object T_Then extends LexerToken
  case object T_Else extends LexerToken
  case object T_While extends LexerToken
  case object T_Do extends LexerToken
  case object T_Repeat extends LexerToken
  case object T_Until extends LexerToken
  case class T_Identifier(var s: String) extends LexerToken
  case class T_Integer(var s: String) extends LexerToken
  case object T_SemiColon extends LexerToken
  case object T_LeftBracket extends LexerToken
  case object T_RightBracket extends LexerToken
  case object T_Equal extends LexerToken
  case object T_LessThan extends LexerToken
  case object T_LessThanEq extends LexerToken
  case object T_EqualComp extends LexerToken
  case object T_GreaterThanEq extends LexerToken
  case object T_GreaterThan extends LexerToken
  case object T_Comma extends LexerToken
  case object T_LeftCurlyBracket extends LexerToken
  case object T_RightCurlyBracket extends LexerToken
  case object T_Assign extends LexerToken
  case object T_Plus extends LexerToken
  case object T_Times extends LexerToken
  case object T_Minus extends LexerToken
  case object T_Div extends LexerToken
  case class T_LexerError(var msg: String) extends LexerToken
}

```

Figure 29 - Defining Tokens in Scala

```

/** Returns token for 'def' keyword */
def `def` = "def" ^^ (_ => T_Def)

/** Returns token for 'if' keyword */
def `if` = "if" ^^ (_ => T_If)

/** Returns token for 'then' keyword */
def `then` = "then" ^^ (_ => T_Then)

```

Figure 30 - Tokenisation of Keywords Using the Parser Combinator Library in Scala

```

/** Returns token for a given legal identifier */
def identifier: Parser[T_Identifier] = {
    "[a-zA-Z][a-zA-Z0-9_]*".r ^^ {s => T_Identifier(s)}
}

/** Returns token for given Integer */
def intLiteral: Parser[T_Integer] = {
    "0|[1-9]+".r ^^ {s => T_Integer(s)}
}

```

Figure 31 - Tokenisation of Identifiers and Integer Literals Using the Parser Combinator Library in Scala

8.1.2 Parser

The power of the parser combinator library becomes more evident in the construction of the parser. Given the context-free grammar in *Figure 32*, the syntax of the tokenised input can be easily checked by defining the syntax given by the context-free grammar using a parser for each rule. *Figure 33* demonstrates how the syntax of a while statement can be checked by defining its syntax within a parser combinator. The parser is made up of four files: `Parser.scala`, `ParserAST.scala`, `SymbolTables.scala` and `Symbol.scala`.

```

PROG → DEC | DEC; PROG
DEC → def ID (VARDEC) = E
VARDEC → ε | VARDECNE
VARDECNE → ID | VARDECNE, ID
ID → (identifiers)
INT → (Integers)
E → INT
  | ID
  | if E COMP E then E else E
  | (E BINOP E)
  | (E; E)
  | while E COMP E do E
  | repeat E until E COMP E
  | ID := E
  | ID(ARGS)
ARGS → ε | ARGSNE
ARGSNE → E | ARGSNE, E
COMP → == | < | > | <= | >=
BINOP → + | - | * | /

```

Figure 32 - Context Free Grammar for Language


```
def `while`: Parser[WhileAST] = {
  (T_While ~ expression ~ comp ~ expression ~ T_Do ~ T_LeftCurlyBracket ~ expression ~ T_RightCurlyBracket) ^^ {
    case a ~ b ~ c ~ d ~ e ~ f ~ g ~ h => new WhileAST(b,c,d,g)
  }
}
```

Figure 33 - Check Syntactical Correctness of While Statement Using Parser Combinator Library in Scala

Much like the tokens produced by the lexer, the ASTs that will be produced by the parser can be trivially defined using case classes as shown in *Figure 34*, allowing for easy pattern matching in the next phases.

```
case class RepeatUntilAST(var body: ExpressionAST, var l: ExpressionAST, var comp: CompAST, var r: ExpressionAST)
case class BinExpAST(var l: ExpressionAST, var binop: BinopAST, var r: ExpressionAST) extends ExpressionAST
case class AssignAST(var x: Int, var e: ExpressionAST) extends ExpressionAST {
```

Figure 34 - Defining ASTs in Scala

8.1.3 Intermediate-Code Generation

The generation of the IR utilises the power of pattern matching in Scala [12]. The ability of generate case classes with different methods within the same file makes defining a rich intermediate representation trivial as depicted by *Figure 35*. This generation phase is made up of two files: TACGen.scala and ThreeAddressInstruction.scala.

```
case class CopyTAC(var arg1: String, var arg2: String) extends ThreeAddressInstruction {
  def getArg1: String = {
    arg1
  }
  def getArg2: String = {
    arg2
  }
}
case class GoToTAC(var label: String) extends ThreeAddressInstruction {
  def setLabel(l: String): Unit = {
    label = l
  }

  def getLabel: String = {
    label
  }
}
```

Figure 35 - Defining an IR in Scala

Pattern matching in Scala allows matching against case classes and therefore each ParserAST can be translated into the relevant sequence of TAC instructions by matching against the case classes defined within ParserAST. *Figure 36* demonstrates the translation of a number of these ASTs, with the eventual collection of TACs being returned by the generator.

```
case RepeatUntilAST(body, l, comp, r) =>
  var lname = "l" + currentInstruction
  appendInstruction(new Label(lname))
  expMatch(body)
  appendInstruction(new IfGoToTAC(compMatch(comp), terminal(l), terminal(r), lname))
case BinExpAST(l, binop, r) =>
  var binE = new BinExpAST(l, binop, r)
  binExpMatch(binE)
case VariableAST(x) =>
  currentVariable = terminal(ast).toInt
case InvokeAST(name, args) =>
  var i = new InvokeAST(name, args)
  procedureCall(i)
}
```

Figure 36 - Translating ASTs in an IR Using Pattern Matching in Scala

8.1.4 Code-Generation

The IR generation phase performs the majority of the complex translation leaving the code-generation phase to output the appropriate x86 assembly code from the TACs it is given. The code-generation phase is made up of one file: Codegen.scala.

Pattern matching is utilised to match against the case classes of ThreeAddressInstruction. The Scala StringBuilder class is used to return the sequence of x86 instructions using the Intel syntax, using the tab character (\t) and newline character (\n) to ensure the output is well formed. The calling conventions for x86 assembly were also followed and this is demonstrated in the translation of FunctionLabel and FunctionExit in *Figure 26*.

```
tac match {  
  case FunctionLabel(name, noOfArgs) =>  
    builder.append(name + "_entry:" + "\tpush ebp\n" +  
      "\tmov ebp, esp\n" +  
      "\tsub esp, " + s"${noOfArgs * 4}" + "\n" +  
      "\tpush ebx\n" +  
      "\tpush ecx\n" +  
      "\tpush edx\n" +  
      "\tpush esi\n" +  
      "\tpush edi\n")  
  case FunctionExit(noOfArgs) =>  
    builder.append("\tmov esp, ebp\n" +  
      "\tpop edi\n" +  
      "\tpop esi\n" +  
      "\tpop edx\n" +  
      "\tpop ecx\n" +  
      "\tpop ebx\n" +  
      "\tpop ebp\n" +  
      "\tret " + s"${noOfArgs * 4}" + "\n")  
}
```

Figure 37 - Generating x86 Assembly Code from an IR Using Pattern Matching in Scala

8.1.5 Testing

All of the testing was performed using the ScalaTest library, more explicitly the FlatSpec [33] that is included within this library. FlatSpec will display each test using a String and this makes evaluating the correctness of each section of the compiler efficient. As seen in *Figure 27*, all tests are self-describing and any failing components can be quickly debugged. The outputs of each component were testing separately using test-driven development. The complete compiler was then testing using a number of input programs.

```

[info] CFICodegenTest:
[info] - should insert labels correctly
[info] - should insert label checks correctly
[info] CompilerTest:
[info] - should compile programs correctly without CFI
[info] - should compile programs correctly with CFI
[info] TACGenTest:
[info] - should translate If Statements correctly
[info] - should translate While Statements correctly
[info] - should translate RepeatUntil Statements correctly
[info] - should translate Assignments correctly
[info] - should translate Binary Expression correctly
[info] - should translate Sequences correctly
[info] - should translate Function invocations correctly
[info] - should translate Programs correctly

```

Figure 38 - Example ScalaTest Output

8.2 Control-Flow Graph Generation

The most challenging part of the project, the generation of the control-flow graph, followed the method for CFG generation as explained in *Compilers: Principles, Techniques and Tools* aka *The Dragon Book* [16]. This phase is made up of three files: `CFGGenerator.scala`, `ControlFlowGraph.scala` and `Block.scala`.

The generation of the CFG is done in three phases. The first of these is to generate the nodes in the graph by splitting the input IR into basic blocks. The block is a simple data type as is depicted in *Figure 39* that contains a list of instructions and a list of the blocks it can pass control-flow to and the blocks from which it is given control-flow.

```

class Block(i: Int) extends ThreeAddressInstruction {

  var blockNumber = i
  var instructions: List[ThreeAddressInstruction] = Nil
  var successors: List[Int] = Nil
  var predecessors: List[Int] = Nil
}

```

Figure 39 - Defining a Block in Scala

The construction of nodes utilises pattern matching, matching each TAC and adding it to the current Block or generating a new Block and adding the fall-through successor. This is easily achieved using the for comprehension in Scala as demonstrated in *Figure 40*.

```
def generateNodes: Unit = {  
  for(i <- instructionList) {  
    instructionMatch(i)  
  }  
}
```

Figure 40 - Generating Nodes Using the For Comprehension in Scala

The difficulty in generating a CFG comes in correctly adding the edges for branches, unlike the fall-through blocks, the block that control-flow would be handed over to is not known until all of the blocks have been produced. This is achieved by performing a second pass over the blocks once the initial nodes and edges have been created. This is achieved by using the enriched IR that included labels. Pattern matching can be used to identify these labels and then add the correct edges as is demonstrated in *Figure 41*.

```

def resolveJump(b: Block): Unit = {
  var i = b.getInstructions.last
  var n = 0

  n = searchLeaders(i)

  if (n != -1)
    b.addSuccessor(n)
    nodes(n).addPredecessor(b.getBlockNumber)
}

def searchLeaders(i: ThreeAddressInstruction): Int = {
  var blockNum = -1
  var labelToMatch = ""

  labelToMatch = branchLabel(i)
  for (n <- nodes) {
    var i1 = n.getInstructions.head
    i1 match {
      case i1: FunctionLabel => {
        if (i1.getName == labelToMatch)
          blockNum = n.getBlockNumber
      }
      case i1: Label => {
        if (i1.getName == labelToMatch)
          blockNum = n.getBlockNumber
      }
      case _ =>
    }
  }

  blockNum
}

```

Figure 41 - Searching Leaders of Blocks in Scala

Much like the node generation, a pass is made over all nodes to ensure that all edges are added/resolved using the for comprehension. *Figure 42* demonstrates this construct.

```
def generateJumpEdges: Unit = {
  for (node <- nodes) {
    resolveJump(node)
  }
}
```

Figure 42 - Resolving Edges Using the For Comprehension in Scala

8.3 Control-Flow Labels

To facilitate the introduction of control-flow labels and checks code generation is performed by traversing the CFG instead of simply pattern matching against the IR. This section is made up of one file: CFICodegen.scala.

```
case CallTAC(name, noOfParam) =>
  builder.append("\tcall " + name + "_entry:\n" +
    "prefetchnta [" + thisCFG.getNodes(currentBlock.getBlockNumber).getLabel + "]\n")
case tag: Label =>
```

Figure 43 - Inserting In-Line Reference Pointers in Scala

Figure 43 demonstrates the insertion of in-line reference pointers into the output string. The unique labels are generated by iterating a hex-value every time a new label is added, this insures that the label is unique and is simple to implement given that Scala allows incrementing of hex-values as you would an Integer. The rest of the code generation is performed using pattern-matching with the ThreeAddressInstructions in each block matched in turn around the production of these reference pointers. Additional checks are then added before each indirect branch with a jump to an error label that prints an error message that ROP has been detected if the checks fail. Figure 44 demonstrates how an indirect branch ("ret") was replaced with a label check.

```

"\tmov ecx, [esp]\n" +
"\tadd esp, " + s"${(noOfArgs+2) * 4}" + "\n" +
"\tcmp [ecx+4], " + thisCFG.getNodes(currentBlock.getBlockNumber).getLabel + "\n" +
"\tjne oferror\n" +
"\tjmp ecx\n" +
```

Figure 44 - Replacing ret with label check

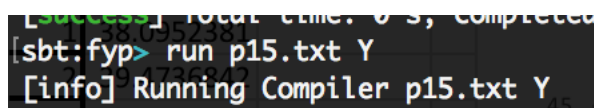
Chapter 9

Testing and Data Analysis

A number of different metrics have been collected during testing to help determine the relative ‘cost’ of implementing CFI. 15 programs were used in this testing and they can be found in *Appendix A*. These programs vary in length and content. The main motivation in the design of these programs was to include a higher volume of indirect branches in some more than other as the concern here is how the additional CFI motivated instructions and the running of these checks effects program compilation and execution. To generate programs with elaborate functionality but very few indirect branches would serve little purpose here as the magnitude of the increase that CFI would put on a program wouldn’t be particularly evident.

7.1 Compile-Time Cost

The data regarding the time taken to compile a given program was collected using the run method of the Scala build tool. Each program was run with and without CFI enabled and the time taken to compile recorded. *Figure 45* demonstrates the final test being run with CFI.



```
[success] Total time: 0 s, completed  
[sbt:fyp> run p15.txt Y  
[info] Running Compiler p15.txt Y
```

Figure 45 - Running Compiler using object Apply() method

The same test was run on all 15 programs, each program passed as a parameter alongside a “Y” or “N” to enable or disable CFI (disabled by default). The data can be seen in *Figure 46*.

Program Number	Compile-Time (milliseconds)	CFI Compile-Time (milliseconds)
1	512	560
2	567	582
3	304	306
4	580	600
5	612	740
6	640	732
7	503	606
8	612	653
9	703	811
10	746	838
11	604	712
12	801	948
13	781	914
14	808	839
15	792	864

Figure 46 - Compile-Time Data

The compile-time cost is defined within the bounds of this report as the percentage increase in the compile-time of a given program. *Figure 47* demonstrates the percentage increase in the compile-time for each program with CFI disabled and then enabled.

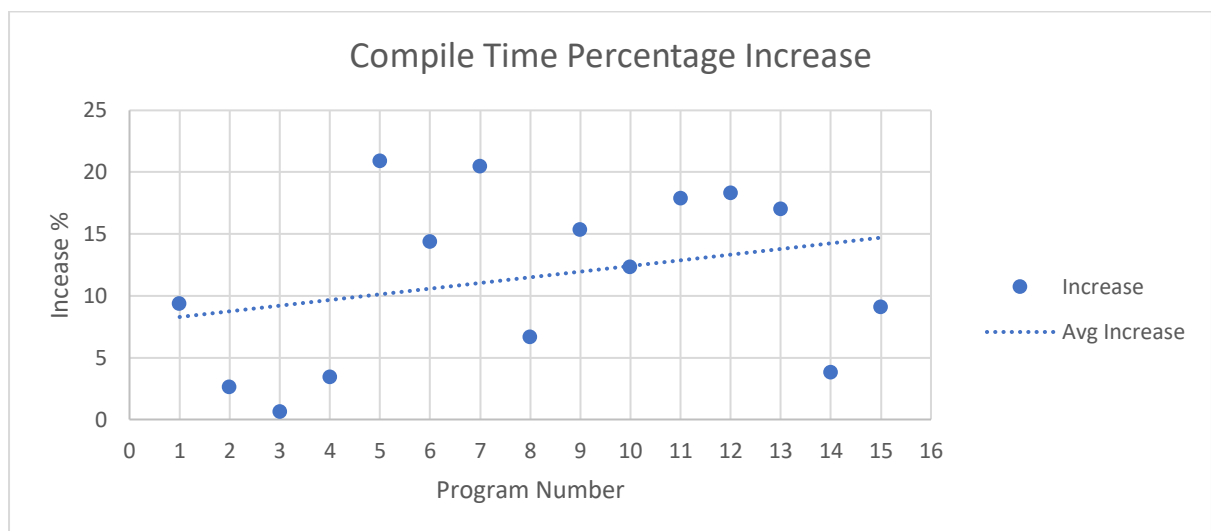


Figure 47 - Compile-Time Percentage Increase

As is evident from the trend line the average percentage increase is around 11.5% a reasonable increase if not considered within the context. However, the increase is only milliseconds and therefore would not be noticeable to a human being, albeit brings some

burden onto the CPU. Moreover, the range is quite erratic and would only really cause concern on an incredibly large input (thousands of lines of code). In reality, the compile-time cost is negligible and is not of real concern here. Each program would only need to be recompiled with CFI once and will then be executable henceforth and therefore is not of particular concern. Of far greater concern is the run-time cost as this will affect every program invocation. It is in that area that comparisons will be made.

7.2 Runtime Cost

The runtime cost is defined within the bounds of this report as encompassing three main areas: the number of instructions that make up the program and therefore have to be executed, the time of execution (seconds) and the cycles performed by the CPU during execution. *Figure 49* demonstrates the percentage increase of the number of instructions produced from CFI being disabled to CFI being enabled.

The data for the number of instructions produced was gathered by analysing the produced output for each compiled source program with and without CFI. This data can be found in *Figure 48*.

Program Number	Instruction Count	CFI Instruction Count
1	21	29
2	38	53
3	70	73
4	57	79
5	74	103
6	128	178
7	130	180
8	142	192
9	175	235
10	187	247
11	186	246
12	186	245
13	224	303
14	264	357
15	259	352

Figure 48 - No. of Instructions Produced Data

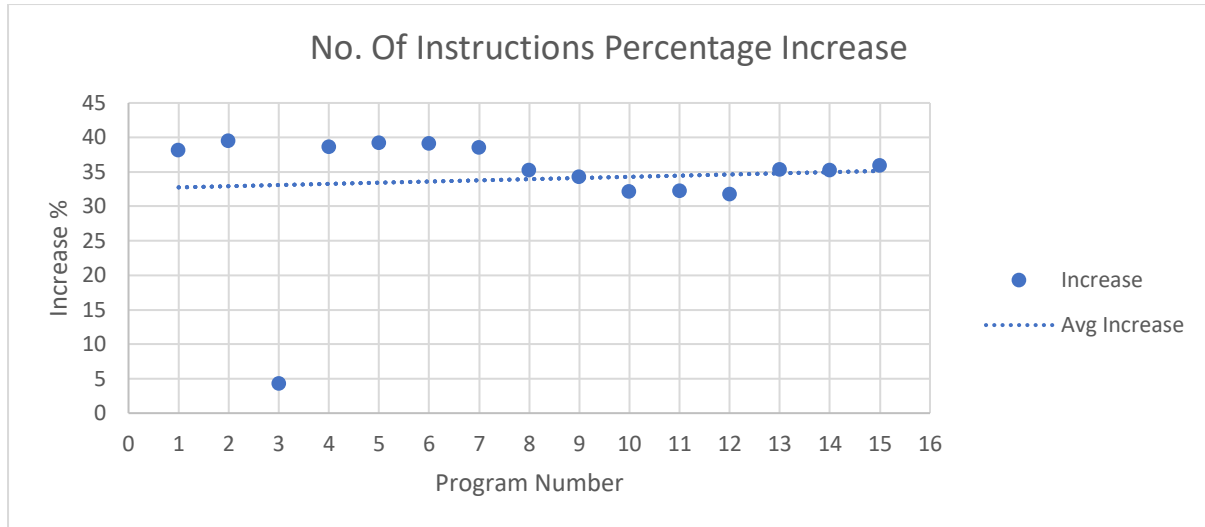


Figure 49 - No. Of Instructions Percentage Increase

It is clear that the number of instructions produced shows a fairly large increase when CFI is enabled. The average increase in instructions produced sits at 33.9%. Given that each indirect branch results in a minimum of 4 additional x86 instructions to implement CFI this increase is not particularly surprising. One would expect similar findings in regard to the cycle counts as most assembly instructions exhaust only a few cycles.

The data regarding the runtime data was captured using the SASM emulator that emulates a NASM assembler. The output program was build and executed and the execution time recorded. SASM [34] was used and not some terminal based benchmark with the motivation of attempting to remove some of the noise that may be caused due to thread scheduling by the OS. This can be seen in Figure 50.

Program Number	Execution-Time (milliseconds)	CFI Execution-Time (milliseconds)
1	30	25
2	22	17
3	11	16
4	36	39
5	42	42
6	37	39
7	32	21
8	26	18
9	52	59
10	46	54
11	51	79
12	68	51
13	72	72
14	66	92
15	84	95

Figure 50 - Runtime Data

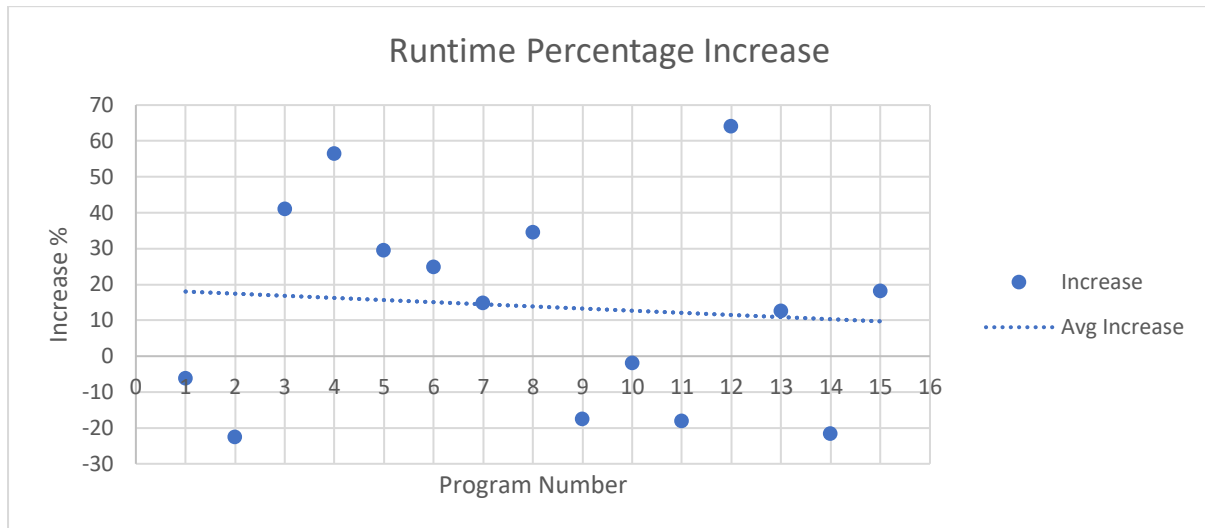


Figure 51 - Runtime Percentage Increase

The percentage increase in terms of runtime could be considered the most important factor in determining the runtime cost given that it will be most noticeable to the user. Depicted by *Figure 51* the average percentage increase was 5.9%, a clearly tolerable increase when considering it is mere milliseconds that will be added onto the runtime of a given program. This bodes well for the viability of CFI.

The cycle count was recorded by executing the compiled program in the Linux terminal after putting it into the correct format (.asm) using SASM and then using NASM in the terminal to make it executable. After each executable the perf stat as part of the Perf library. This was performed with no other programs running in an attempt to avoid noise in this reading (however with modern CPUs this is very difficult, so the results will show some perhaps unnatural variation). This data can be seen in *Figure 52*.

Program Number	Cycle Count	CFI Cycle Count
1	20	32
2	38	53
3	72	74
4	56	81
5	72	107
6	123	171
7	134	179
8	144	196
9	176	231
10	184	251
11	186	246
12	182	251
13	226	336
14	266	352
15	264	368

Figure 52 - Cycle Count Data

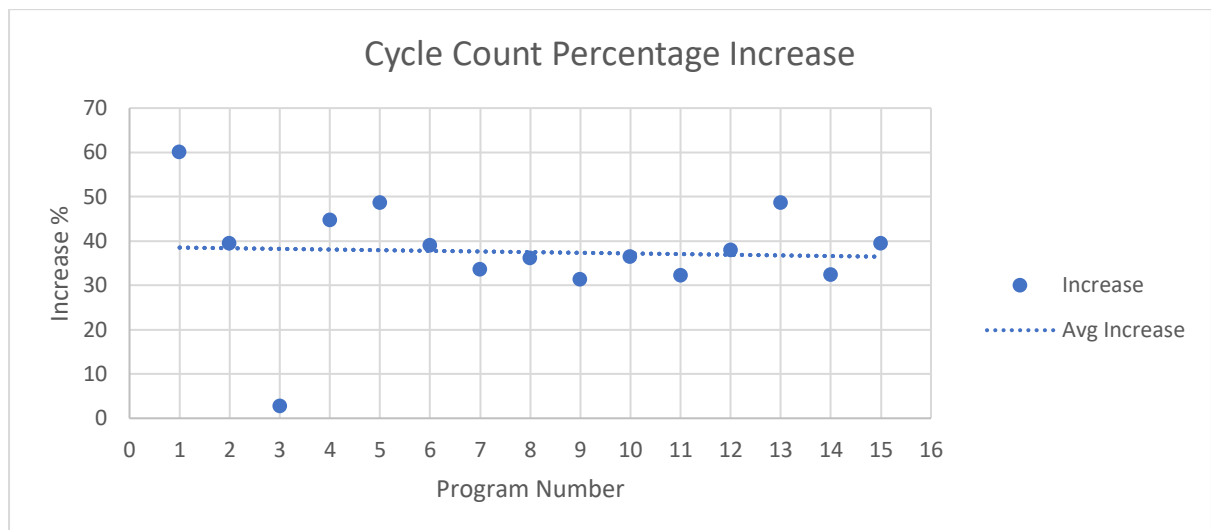


Figure 53 - Cycle Count Percentage Increase

The percentage increase in cycle count is fairly consistent with the percentage increase of instructions produced, sitting at 37.5%, as can be seen when comparing *Figure 53* with *Figure 49*.

7.2.1 kBouncer Comparison

Given its implementation as a kernel module kBouncer can be benchmarked using a number of pre-existing OS programs and its % overhead, that could be referred to as its runtime cost, can be calculated by looking at the average time of an LBR checking and using this alongside the number of system-calls to generalise its execution time.

Type	Times			Calls/ Returns	Syscalls	False Positives	Overhead	
	real	user	sys				ms	%
wmplayer	30.73	0.37	0.21	30.8M	194K	0*	33.8	6
iexplore	7.24	0.06	0.04	1.5M	31K	0	5.4	5
reader	4.11	1.32	0.24	35.3M	107K	0	18.7	1

Figure 54 - kBouncer Benchmarks [23]

Figure 54 presents the data gathered from running three pre-existing Windows application with kBouncer. The average increase in runtime cost is 4% across the application that vary in their complexity. Compared with the 5.9% average increase for the CFI implementation kBouncer would seem to be more efficient. However, given the small data set used in kBouncer evaluation this comparison isn't wholly fair and a larger data set may prove the overhead to be very similar to CFI. kBouncer brings the advantage of using LBR which itself brings very little overhead as it is hardware-based. Given the abundance of checks performed by kBouncer however, its overhead begins to come close to the user-level execution of CFI. This would suggest that CFI is viable as a defence mechanism, especially if it proves to be more effective in ROP detection.

7.2.2 ROPecker Comparison

Given its implementation as a kernel module ROPecker can be bench-marked using the SPEC CPU 2006 benchmarks, an industry standard for benchmarking. Unfortunately, without recompiling all of the native programs in an OS this cannot be achieved for CFI. However, a comparison can still be made as to the runtime cost.

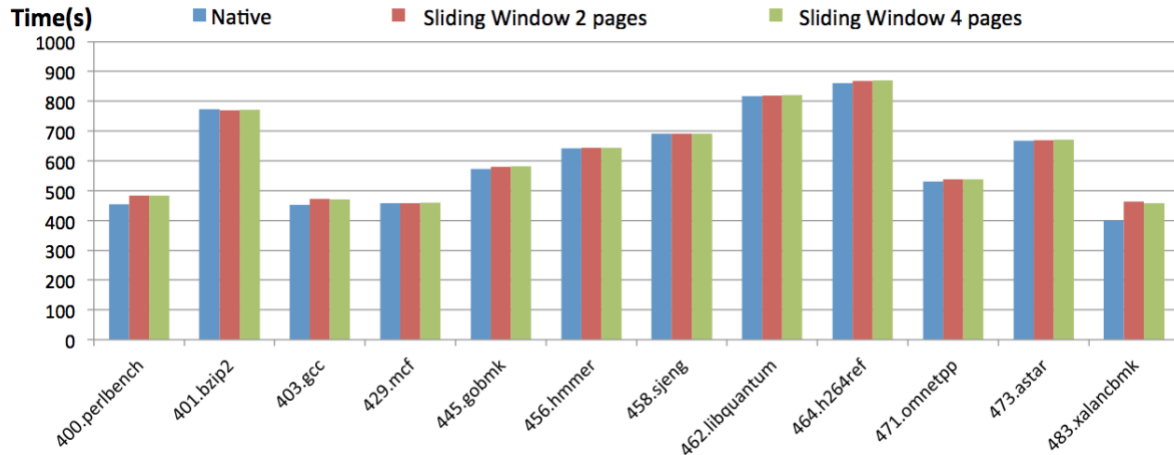


Figure 55 - ROPEcker SPEC Benchmarks [26]

Figure 5 displays the results of running a number of benchmarks from the SPEC suite. The average increase in runtime cost was found to be 2.6% across these benchmarks. This increase is more favourable than that of this CFI implementation which show an average increase in runtime cost in terms of execution time of 5.9%. This difference can be attributed to the fact that the LBR is a hardware-based solution and therefore carries very little overhead against the user-level instruction sequences produced by CFI. In addition, given that ROPEcker operates within the kernel space and not user space like CFI execution is likely to be faster given that it is slightly more low-level. With this in mind, the increase in number of instructions produced and cycle count clearly has a detrimental effect on the runtime cost. This would suggest that there are perhaps more viable ROP-mitigation mechanism than CFI that use LBR. However, this may become clearer if there is a distinct difference in the effectiveness of the ROP detection.

7.3 Attacks Foiled

Gauging the percentage of ROP-attacks prevented on each program as each program itself will have different useful ROP-chains. Instead this will be discussed abstractly. The source language used in this project is not particularly complex and therefore does not translate into a dense set of instructions. Given this, and with uniqueness of IDs, the NX-bit and DEP, the CFI implementation proved to be very robust and through initial testing during development and attempts to circumvent it post (albeit not a full ROP-exploit given the attack is assumed controlled). The same can be said for kBouncer which mitigated all return-based attacks it faced. Where ROPecker perhaps exceed over the two is in its detection of gadget chains that feature jumps and not returns. These kinds of attacks, more advanced than a standard ROP attack, use arbitrary jumps in the code to simulate returns. This however requires a lot more knowledge of the location of code in memory. Due to its ability to detect 'ROP-like' behaviour kBouncer benefits here.

7.4 Viability

Control-Flow Integrity has proven to be an effective defence against ROP. The less important compile time cost and poignant runtime costs are certainly tolerable for execution on modern day CPUs. The viability of CFI is damaged by the existence of this compile time metric. The complexity of the programs that are often targeted on modern systems (Internet Explorer, Silverlight etc...) are massive programs installed on millions of machines around the world. To recompile the programs with the additional code and then push this update out is not viable given the ever-shifting nature of the cyber security arms race. Kernel-module based techniques, that analyse and run-alongside existing programs such as ROPecker certainly appear to be more viable given the need for pragmatism.

Chapter 8

Evaluation

This section of the report will focus on evaluating this project as whole. This evaluation will look at the project aims and establish if they were achieved and if so evaluate the quality of each achievement. This evaluation will focus on two parts in particularly:

1. The compiler and CFI implementation
2. The evaluation of the relative 'cost' of CFI

8.1 The Implementation

Two of the main objectives for the implementation were achieved in that the compiler with optional CFI generation was successfully implemented and functions as expected. Only one objective was not achieved: implement an alternative to CFI, perf counters.

This objective was not met mainly due to the scope of this report. Attempts were made to use the last-branch register in a similar fashion to CFI, to provide a point of comparison. However, the information stored in the LBR is wholly runtime dependent, and although symbolic offsets could in theory be used for all locations and checks implemented based on this and whatever values the LBRs may hold at runtime, this implementation was deemed impractical and clearly an inefficient solution to the problem. It was therefore concluded that no further time would be invested into this as it was unlikely to bear useful results to aid in the evaluation of CFI. The use of the LBRs is more feasible in kernel modules that can intervene at runtime as demonstrated by the given examples of kBouncer and ROPecker.

The main area for improvement in terms of this would be to leverage LBR in a kernel module of some description to provide a basis for evaluation using the same dataset for both CFI and the LBR. However, as is clear from the literature regarding kBouncer and ROPecker this can quickly become a project within itself. With a larger timeframe in which to work on the project this may become a feasible task.

8.2 Cost Evaluation

Three of the main objectives were achieved in this area as expected with the 'cost' of implementing CFI at both compile-time and runtime being evaluated with data gathered about the implemented compiler and the programs it produced. Discussion around the viability of CFI was also achieved. The objectives that were not achieved, or to be more accurate not achieved in the expected fashion was again in the area surrounding the sue of LBRs.

Given that an implementation using LBR was not achieved calculating the direct 'cost' of implementing it was difficult. However, effective comparison was achieved by considering data from the literature regarding the given examples kBouncer and ROPecker.

Although this gave some ground for comparison and generated discussion, to provide an accurate comparison of a CFI implementation vs an LBR implementation all of the tests should be performed on the same data set so direct comparison can be drawn. This is the main area for improvement in regard to the project as a whole.

Conclusion

This project has provided an insight into compilers, control-flow, Return-Orientated Programming (ROP) and ROP mitigation techniques in the form of Control-Flow Integrity (CFI) and the use of Last-Branch Registers (LBR). CFI has been shown to be a robust and not wholly viable defence mechanism in mitigating ROP when compiling from our simple source language.

It is clear that CFI has some place in the defence against ROP, however the hassle of having to recompile all current programs does damage its feasibility in the real world. Implemented kernel modules like kBouncer and ROPecker get around this and perhaps provide a more practical approach to defence against ROP than CFI.

The main objectives regarding this project have been achieved given that CFI was successfully implemented into a compiler and a basis from which an evaluation of the viability of CFI could be evaluated was established.

In hindsight, an alternative methodology that would have helped generate a better discussion of CFI would have been to modify an existing compiler. With the implementation of the compiler removed from the project schedule, greater time could have been invested to successfully implement an implementation that utilises LBR and a more direct comparison could have been made between CFI and LBR using data from produced from the same inputs.

Bibliography

- [1] Intel. [Online]. Available: <https://www.intel.co.uk/content/www/uk/en/silicon-innovations/moores-law-technology.html>. [Accessed 20 04 2018].
- [2] H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," in *ACM CCS 2007*, ACM Press, 2007..
- [3] M. Abadi, M. Budiu, U. Erlingsson and J. Ligatti, "Control-Flow Integrity Principles, Implementations, and Applications," *Journal ACM Transactions on Information and System Security*, vol. 13, no. 1, p. Article 4, 2009.
- [4] Intel, "Control-Flow Enforcement Technology Preview," 06 2017. [Online]. Available: <https://software.intel.com/sites/default/files/managed/4d/2a/control-flow-enforcement-technology-preview.pdf>. [Accessed 02 11 2017].
- [5] G. Anthes, "Timeline: A brief history of the x86 microprocessor," [Online]. Available: <https://www.computerworld.com/article/2535019/computer-hardware/timeline--a-brief-history-of-the-x86-microprocessor.html>. [Accessed 18 04 2018].
- [6] D. Evans. [Online]. Available: <http://www.cs.virginia.edu/~evans/cs216/guides/x86.html>. [Accessed 20 04 2018].
- [7] D. E. Quentin Carbonneaux. [Online]. Available: <http://flint.cs.yale.edu/cs421/papers/x86-asm/asm.html>. [Accessed 19 04 2018].
- [8] Intel, "Intel® 64 and IA-32 Architectures Software Developer Manuals," [Online]. Available: <https://software.intel.com/en-us/articles/intel-sdm>. [Accessed 22 04 2018].
- [9] Oracle, "Java Software," [Online]. Available: <https://www.oracle.com/uk/java/index.html>. [Accessed 11 03 2018].
- [10] Apple, "Swift 4," [Online]. Available: <https://developer.apple.com/swift/>. [Accessed 20 03 2018].
- [11] Haskell, "Haskell," [Online]. Available: <https://www.haskell.org/>.
- [12] Scala, "The Scala Programming Language," [Online]. Available: <https://www.scala-lang.org/>. [Accessed 22 04 2018].
- [13] ARM, "Arm," [Online]. Available: <https://www.arm.com/>. [Accessed 10 03 2018].
- [14] D. M. Berger, "Compilers and Computer Architecture," [Online]. Available: <https://studysdirect.sussex.ac.uk/course/view.php?id=27392&topic=3>. [Accessed 28 04 2018].
- [15] M. S. L. R. S. J. D. U. Alfred V. Aho, *Compilers: Principles, Techniques, and Tools*, Pearson Education, Inc, 1986, p. 367.
- [16] M. S. L. R. S. J. D. U. Alfred V. Aho, *Compilers: Principles, Techniques, and Tools*, Pearson Education, Inc, 1986, pp. 529, 530.
- [17] Microsoft, "A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003," [Online]. Available: <https://support.microsoft.com/en-us/help/875352/a-detailed-description-of-the-data-execution-prevention-dep-feature-in>. [Accessed 20 04 2018].

- [18] K. Kubicki, "A Bit About the NX Bit;Virus Protection Woes," [Online]. Available: <https://www.anandtech.com/show/1507>. [Accessed 19 04 2018].
- [19] Microsoft, "Control Flow Guard," [Online]. Available: [https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065\(v=vs.85\).aspx](https://msdn.microsoft.com/en-us/library/windows/desktop/mt637065(v=vs.85).aspx). [Accessed 29 04 2018].
- [20] S. Krahmer, "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique," [Online]. Available: <https://users.suse.com/~krahmer/no-nx.pdf>. [Accessed 18 04 2018].
- [21] A. G. PROJECT, "The GNU C Library (glibc)," [Online]. Available: <https://www.gnu.org/software/libc/>. [Accessed 21 04 2018].
- [22] Intel, "Intel® 64 and IA-32 Architectures Software Developer's Manual," 10 2017. [Online]. Available: <https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf>. [Accessed 2 11 2017].
- [23] V. Pappas, "Defending against Return-Oriented Programming," 2015. [Online]. Available: https://www1.cs.columbia.edu/~angelos/Papers/theses/vpappas_thesis.pdf. [Accessed 2 11 2017].
- [24] Perf, "Linux Kernel Profiling with Perf," [Online]. Available: <https://perf.wiki.kernel.org/index.php/Tutorial>. [Accessed 29 04 2018].
- [25] A. Kleen, "An introduction to last branch records," 23 03 2016. [Online]. Available: <https://lwn.net/Articles/680985/>. [Accessed 29 04 2018].
- [26] Z. Z. M. Y. D. X. D. R. H. Cheng Yueqiang, "ROPecker: A Generic and Practical Approach For Defending Against ROP Attack," [Online]. Available: http://ink.library.smu.edu.sg/cgi/viewcontent.cgi?article=2972&context=sis_research. [Accessed 29 04 2018].
- [27] Scala, "Scala," [Online]. Available: <https://www.scala-lang.org/>.
- [28] Scala, "Scala Parser Combinators," [Online]. Available: <https://github.com/scala/scala-parser-combinators>.
- [29] B. Venners and Artima, "Scala Test," [Online]. Available: <http://www.scalatest.org/>.
- [30] G. Inc., "GitHub," 2017. [Online]. Available: <https://github.com/>.
- [31] BCS, "BCS The Chartered Institute for IT," 2017. [Online]. Available: <http://www.bcs.org/>.
- [32] BCS, "Code of Conduct," 2017. [Online]. Available: <http://www.bcs.org/category/6030>.
- [33] ScalaTest, "FlatSpec," [Online]. Available: <http://doc.scalatest.org/1.8/org/scalatest/FlatSpec.html>. [Accessed 24 04 2018].
- [34] D. Manushin, "SASM," [Online]. Available: <https://dman95.github.io/SASM/english.html>. [Accessed 20 04 2018].
- [35] R. Nilsson, "Scala Check," [Online]. Available: <https://www.scalacheck.org/>.
- [36] O. Source, "Git Hooks," [Online]. Available: <https://git-scm.com/book/gr/v2/Customizing-Git-Git-Hooks>.
- [37] M. Odersky, "Functional Programming Principles in Scala," [Online]. Available: <https://www.coursera.org/learn/progfun1>.

[38] D. Manushin, "SASM," 2017. [Online]. Available:
<https://dman95.github.io/SASM/english.html>.

[39] C. Ltd., "Ubuntu," 2017. [Online]. Available: <https://www.ubuntu.com/>.

Appendix A

p1.txt

```
def main() = {  
    x := 1  
}
```

p2.txt

```
def main() = {  
    branch1()  
}  
def branch1() = {  
    x := 2  
}
```

p3.txt

```
def main() = {  
    branch1()  
}  
def branch1() = {  
    (1+1)  
    (2+2)  
    (3+3)  
    (4+4)  
    (5+5)  
    (6+6)  
    (7+7)  
}
```

p4.txt

```
def main() = {  
    branch1()  
}  
def branch1() = {  
    branch2()  
}  
def branch2() = {  
    (3-2)  
}
```

p5.txt

```
def main() = {  
  branch1()  
}  
def branch1() = {  
  branch2()  
}  
def branch2() = {  
  branch3()  
}  
def branch3() = {  
  (1+1)  
}
```

p6.txt

```
def main() = {  
  branch1()  
}  
def branch1() = {  
  branch2()  
}  
def branch2() = {  
  branch3()  
}  
def branch3() = {  
  branch4()  
}  
def branch4() = {  
  (branch5();branch6())  
}  
def branch5() = {  
  (1*3)  
}  
def branch6() = {  
  (2-1)  
}
```


p7.txt

```
def main() = {  
  branch1()  
}  
def branch1() = {  
  branch2()  
}  
def branch2() = {  
  branch3()  
}  
def branch3() = {  
  branch4()  
}  
def branch4() = {  
  (branch5(2);branch6())  
}  
def branch5(a) = {  
  (1*3)  
  x := a  
}  
def branch6() = {  
  (2-1)  
}
```

p8.txt

```
def main() = {  
  branch1()  
}  
def branch1() = {  
  branch2()  
}  
def branch2() = {  
  branch3()  
}  
def branch3() = {  
  branch4()  
}  
def branch4() = {  
  (branch5(2);branch6())  
}  
def branch5(a) = {  
  (1*3)  
  while a < 4 do {  
    a := (a+1)  
  }  
}
```

```

    }
}
def branch6() = {
    (2-1)
}

```

p9.txt

```

def main() = {
    branch1()
}
def branch1() = {
    branch2()
}
def branch2() = {
    branch3()
}
def branch3() = {
    branch4()
}
def branch4() = {
    branch5(2)
}
def branch5(a) = {
    (1*3)
    while a < 4 do {
        x := (a-1)
        branch7(x)
        a := (a+1)
    }
}
def branch6() = {
    (2-1)
}
def branch7(b) = {
    if b > 2 then {
        branch6()
    } else {
        (1+1)
    }
}

```

p10.txt

```
def main() = {  
  branch1()  
}  
def branch1() = {  
  branch2()  
}  
def branch2() = {  
  branch3()  
}  
def branch3() = {  
  branch4()  
}  
def branch4() = {  
  branch5(2)  
}  
def branch5(a) = {  
  (1*3)  
  while a < 4 do {  
    x := (a-1)  
    branch7(x)  
    a := (a+1)  
  }  
}  
def branch6() = {  
  (2-1)  
}  
def branch7(b) = {  
  if b > 2 then {  
    while b < 2 do {  
      b := (b+1)  
    }  
  } else {  
    (1+1)  
  }  
}
```

p11.txt

```
def main() = {  
  branch1()  
}  
def branch1() = {  
  branch2()  
}  
def branch2() = {  
  branch3()  
}  
def branch3() = {  
  branch4()  
}  
def branch4() = {  
  branch5(2)  
}  
def branch5(a) = {  
  (1*3)  
  while a < 4 do {  
    x := (a-1)  
    branch7(x)  
    a := (a+1)  
  }  
}  
def branch6() = {  
  (2-1)  
}  
def branch7(b) = {  
  if b >= 2 then {  
    while b == 2 do {  
      branch5(3)  
    }  
  } else {  
    (1+1)  
  }  
}
```

p12.txt

```
def main() = {
  branch1()
}
def branch1() = {
  branch2()
}
def branch2() = {
  branch3()
}
def branch3() = {
  branch4()
}
def branch4() = {
  branch5(2)
}
def branch5(a) = {
  (1*3)
  while a < 4 do {
    x := (a-1)
    branch7(x)
    a := (a+1)
  }
}
def branch6() = {
  (2-1)
}
def branch7(b) = {
  if b > 2 then {
    while b < 2 do {
      branch8(3)
    }
  } else {
    (1+1)
  }
}
def branch8(c) = {
  while c > 1
    c := (c-1)
}
```

p13.txt

```
def main() = {
  branch1()
}
def branch1() = {
  branch2()
}
def branch2() = {
  branch3()
}
def branch3() = {
  branch4()
}
def branch4() = {
  (branch5(2);branch6())
}
def branch5(a) = {
  (1*3)
  while a < 4 do {
    a := (a+1)
  }
}
def branch6() = {
  (2-1)
}
def branch7() = {
  branch8()
}
def branch8() = {
  branch9()
}

def branch9() = {
  branch10(4)
}

def branch10(d) = {
  if d == 4 then {
    (5+1)
  } else {
    (5-1)
  }
}
```

p14.txt

```
def main() = {
  branch1()
}
def branch1() = {
  branch2()
}
def branch2() = {
  branch3()
}
def branch3() = {
  branch4()
}
def branch4() = {
  (branch5(2);branch6())
}
def branch5(a) = {
  (1*3)
  while a < 4 do {
    a := (a+1)
  }
}
def branch6() = {
  (2-1)
}
def branch7() = {
  branch8()
}
def branch8() = {
  branch9()
}

def branch9() = {
  branch10(4)
}

def branch10(d) = {
  if d == 4 then {
    branch11(4)
  } else {
    branch12(5)
  }
}
```

```
def branch11(e) = {  
  x := e  
  x := (x+1)  
}  
def branch12(f) = {  
  y := f  
  y := (y-1)  
}
```


p15.txt

```
def main() = {
  branch1()
}
def branch1() = {
  branch2()
}
def branch2() = {
  branch3()
}
def branch3() = {
  branch4()
}
def branch4() = {
  (branch5(2);branch6())
}
def branch5(a) = {
  (1*3)
  while a < 4 do {
    a := (a+1)
  }
}
def branch6() = {
  (2-1)
}
def branch7() = {
  branch8()
}
def branch8() = {
  branch9()
}

def branch9() = {
  branch10(4)
}

def branch10(d) = {
  if d == 4 then {
    branch11(4)
  } else {
    branch12(5)
  }
}
```

```
def branch11(e) = {  
  x := e  
  branch5(x)  
}  
def branch12(f) = {  
  branch11(f)  
}
```

Appendix B

Log

Completed Date	Task
July	Complete Coursera Course
August	Read ROP proposal paper
August	Read CFI proposal paper
August	Read LBR paper
September	Familiarise with Intel x86 assembly
October	Create and test code generation
October	Create and test lexer
October	Create and test parser
November	Interim report
November	Test Compiler
January	Create and test CFGGenerator
February	Create and test CFI
March-April	Attempt LBR
April	Submit draft
April	Submit final report