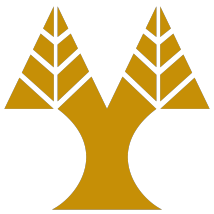
Thesis Dissertation

MBDISASS: ANALYSIS OF MIXED BINARIES

UNIVERSITY OF CYPRUS



COMPUTER SCIENCE DEPARTMENT

May 2018

UNIVERSITY OF CYPRUS
COMPUTER SCIENCE DEPARTMENT

mbDisass: Analysis of Mixed Binaries

Michalis Papaevripides

Supervisor
Dr. Elias Athanasopoulos

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

May 2018

Acknowledgements

I would like to express my sincere gratitude to my thesis advisor Dr. Elias Athanasopoulos for guiding me throughout my research.

I am also grateful for my family and friends who provided me with support and continuous encouragement throughout the process of the thesis.

Abstract

Static binary analysis is vital for the reverse engineering community. Binary instrumentation and vulnerability search at the binary level rely on the good understanding of the program's control flow using static analysis tools. In addition, as control flow hijacking is one of the most widely used attacks for software exploitation performed by attackers, now is more important than ever to make sure modern software won't constitute easy targets for the attackers.

The core practice used by attackers in order to achieve control flow hijacking is by tampering control data, namely data that are used by the programs' indirect branches in order to achieve jumps determined on the runtime rather at the compilation time. For example indirect branches are utilized in C through function pointers and in C++ through VTable pointers in order to achieve dynamic dispatching.

Understanding a program's flow becomes even more difficult not only because modern software gets more complex but also because many programming languages get intermixed together. Binaries with intermixed programming languages could pose a great danger in the future since languages like Rust that doesn't enforce memory safety at runtime could be used for tampering control data and consequently achieving control flow hijacking.

We propose mbDisass, a static binary analysis tool for mixed binaries. mbDisass's analysis consists of two phases. The first phase is responsible for disassembling and storing the functions of the disassembled executable in a database. The second phase is responsible for loading the disassembled functions from the previous phase, finding all the indirect branches contained in the functions and finally classifying each indirect branch to a programming language among C,C++ and Rust.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Static binary analysis is vital for the reverse engineering community. As software gets more complex, a need for powerful static binary analysis tools arise. Static binary analysis help the reverse engineering community to understand how a program works without the need of the source code. In addition binary instrumentation [12] and vulnerably search [11] at the binary level rely on the good understanding of the program's control flow.

Software attacks that alternate a program's control flow have been widely used for a long time. Such exploits are divided into two categories, forward edge attacks and backward edge attacks in regards to the Control Flow Graph (CFG).In the last few years there has been an increase number of software attacks using the forward edge rather the backward edge. This is due to the fact that the computer security research community successfully managed to protect the backward edge by securing the return addresses and other vital stack based data using stack cookies and other hardware methods [8, 14]. However forward edge attacks are still an unsolved problem as there is not an efficient defence yet that prevents these attacks. Control flow hijacking is one of the most widely used attack in software exploitation . In order to achieve control flow hijacking, attackers tamper control data to divert a program from its intended flow. Function pointers used for target resolving at runtime in languages like C and C++ and virtual function table pointers called VTable pointers for dynamic dispatching constitute the main targets to achieve control flow hijacking [25].

Modern commercial software mix various programming languages to achieve their targets. For example Mozilla Firefox intermixes C++ and Rust [7]. As a result, binaries

originated from different programming languages are running simultaneously. Consequently indirect branches originated from different programming languages get mixed together. For this reason we built mbDisass, a static binary analysis tool that disassembles an executable, finds the indirect branches included in a binary file and classifies them to a programming language using heuristic rules. The programming languages that the heuristic rules search for are C, C++ and Rust.

Rust statically enforces memory safety at compilation time for efficiency reasons in contrast with other programming languages which enforce memory safety at runtime [20]. The previous statement in combination with the fact that Rust binaries get mixed with binaries originated from other programming languages raises potential exploitation threads. For example we believe that if an attacker tampers the control data of a programming language which enforces memory safety at runtime like a VTable pointer of C++ and uses that exploit to manipulate the control data coming from Rust, could result in catastrophic consequences as this would result in cancelling the safety guarantees of Rust.

Despite the fact that there are many defences that protect against software exploitation [2, 8, 14, 22, 24] there isn't an efficient and effective defence against control flow hijacking. Even with the core defence against it which is Control Flow Integrity [1], software is still vulnerable to control flow hijacking [6,9,10,23]. For this reason, is crucial to find the indirect branches contained in a binary and recognize their origin programming language. This procedure is very important because it could help the reverse engineering community to identify early on various vulnerabilities that could pose a thread in the future. In addition identifying the indirect branches in the binary could be very helpful for better understanding the control flow of a program.

The above effort has some serious challenges. Firstly it is well known that disassembly is an open problem [3]. Inaccuracies in disassembly could result in poor static binary analysis, consequently an awful indirect branch detection. Moreover, with an extension to the disassembly, accurate function detection constitutes a crucial need regarding the indirect branches resolving. However function detection tools face two main problems, function start detection and function boundary detection, these two can be a difficult task as they face challenging cases like padding, unreachable code and tail calls [4]. Finally, classifying indirect branches to a programming language is a difficult task. The reason behind this is because there isn't a single indirect branch pattern at the binary level that could determine the origin programming language.

mbDisass is divided into two main sections, the Disassembler and the Heuristic. The

disassembler loads the executable's binary and disassembles it in batches to generate an accurate disassembly. Afterwards the disassembler creates data structures containing the disassembly and the functions' entry points. Lastly, the heuristic finds the indirect branches and uses a number of heuristic rules to classify them in each of the three programming languages, C, C++ and Rust. The heuristic rules try to identify patterns that show up at the binary level from each of the three programming languages mentioned before.

We present mbDisass, a static binary analysis tool for mixed binaries that locates and classifies indirect branches to a programming language. It provides the following contributions:

- A mixed binary disassembler.
- Successfully finds over indirect branches contained in an executable/library with a high accuracy.
- Extensibility for analysing indirect branches from programming languages other than C,C++ and Rust.
- Fast indirect branches analysis.

Chapter 2

Background

2.1 Disassembly

Disassembly is the translation of machine language into assembly language which is a human-readable format and constitutes the foundation for static binary instrumentation research. However it is well known that is an unsolved problem, as a result, static analysis becomes even more challenging. Complex constructs like in-line data and padding result most of the times to an inaccurate disassembly [3]. Accurate disassembly is crucial for binary based research as it enables binary-level vulnerability search, binary-level anti-exploitation systems and generally program analysis when source code is not available.

```
0x55 0x48 0x8b 0x05 0xb8 0x13 0x00 0x00
```

a: Binary code in hex

```
push    rbp
mov rax, qword ptr [rip + 0x13b8]
```

b: Disassembly

Listing 2.1.1: Binary code in hexadecimal notation to disassembly example. As shown in the Listing, the left one 2.1.1a represents binary code in hexadecimal notation. As shown from the right Listing 2.1.1b, after disassembling the binary code we get a human readable representation which makes it feasible for static analysing an executable.

2.2 Binary Static Analysis

Binary Static Analysis is the act of analysing the binary of a program without executing it. Binary Static Analysis is one of the key elements for human reverse engineering. Not only it is crucial for understanding complex binary code but also vital for discovering vulnerabilities that could pose a threat in the future.

2.3 Memory Safety Exploits

Memory corruption bugs constitute the main source of software exploitation. To achieve a memory safety exploit, the attacker can either dereference an out-of-bounds pointer which is called spatial error, or dereference a dangling pointer which is called a temporal error [25]. Using dangling pointers namely those that point to a deleted objects and invalid pointers namely those that point to an invalid address are two core memory safety exploits that are used by attacker to take control of a program and execute malicious

code. Dangling pointers are mainly used in virtual table hijacking and invalid pointers in function pointer hijacking.

2.4 Defences against Memory Safety Exploits

Memory Safety Exploitation has bother security research community for a long time because by using a bug, attackers can gain malicious code execution over a program, this might result in the diversion of the control flow with unintended code execution. Some of the most widely deployed defences against this issue follow below.

2.4.1 Data Execution Prevention

Data Execution Prevention (DEP) [2] combines both software and hardware technologies in order to prevent input data execution in regions like the stack and the heap. Basically it determines the executable regions and non executables regions in order to prevent memory safety exploits, for example heap spraying and stack shell code injection are no longer immediate threads without first surpassing the DEP. Nonetheless Return Oriented Programming(ROP) [6,9,10,23] constitutes on of the main workaround against DEP. Attackers take advantage of small chunks of code ending with "ret", called gadgets. When correctly placed data in the stack gets combined with a chain of gadgets then the program diverts from its indented control flow and as a result, malicious code execution is achieved. An example can be seen in Figure 2.2. Consequently Control Flow Hijacking still occurs with DEP in place.

2.4.2 Address Space Layout Randomization

Address Space Layout Randomization (ASLR) [22, 24] is considered a very effective defence against memory safety exploits as it randomizes the processes loading address space layout. The process loading address is a vital information for developing software exploits, thus without it, attackers are unable to develop any exploits. However this defence is weak against Information Leaks. Therefore Memory Safety Exploits can still occur with ASLR in place.

2.4.3 Stack Cookies

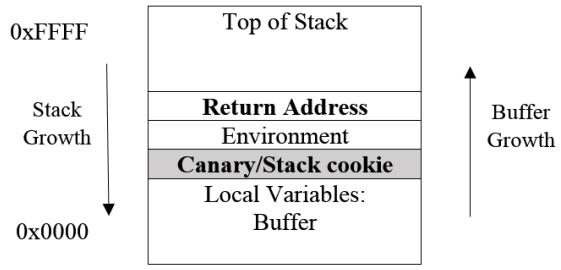


Figure 2.1: Stack Canary example. The above figure illustrates an example of how a stack cookie works. Basically as the buffer grows up, in case it exceeds its fixed size then the stack cookie prevents it from overwriting the return address.

Stack cookies [8, 14] (Figure 2.1) placed in the stack to prevent overwriting control data like return addresses using a linear overflow. When a program writes more data to a buffer than its fixed allocated size, its called Buffer Overflow and it used to be a major exploitation thread in the past until the deployment of stack cookies. Nevertheless, because of the use of stack canaries which mainly protect the backward edge, attackers shifted their focus to the forward edge exploitation. The previous statement in combination with the fact that information leaks can compromise the protection of stack cookies, results programs to still be vulnerable to software exploitation even with stack cookies in place.

As we can conclude, despite the promising defences that are widely deployed, we still face software exploitation threats that manage to take control of our software.

2.5 Control flow hijacking

A common memory-corruption method used in software exploitation these days is Control Flow Hijacking where an attacker tries to divert the flow of a program by overwriting control data. Control flow is divided into two categories, Backward edge and Forward edge. Backward edge is used when a callee function returns to the caller function address and Forward edge is used by functions pointers and virtual function tables pointers used for dynamic dispatching. Even though several widely adopted techniques have been used to protect the backward edge, protecting the forward edge remains an open problem. This happens because Control Flow Integrity(CFI), the technique used for protecting forward edge has high overhead and impractical assumptions, as a result it hasn't been integrated in production compilers. A simplified example of Control Flow Hijacking can be seen

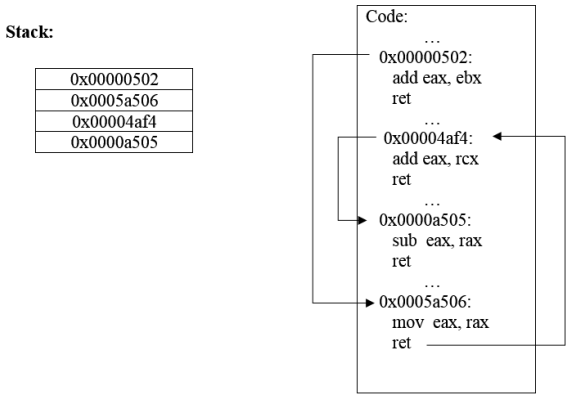


Figure 2.2: Return Oriented Programming(ROP) example. The above figure illustrates a simplified example of a ROP attack. As it can be seen, the attacker uses small pieces of binary code ending with "return" shown at the right and there addresses are carefully placed on the stack shown on the left. As a result when a small piece ends with a "return", the program the proceeds by jumping on the next addresses placed in the stack.

from the Figure 2.3. In this specific occasion, the register "%rax" holds the address of the function that the program is going to call next. If an attacker manages to temper any control data, namely the register "%rax" and replace its target address with an address which points to a malicious code, then the attacker manages to gain control of the program and gains arbitrary code execution.

2.6 Control Flow Integrity

Control Flow Integrity (CFI) [1] (Listing 2.6.1) is the main protection method against Control Flow Hijacking aside from ASLR, DEP and stack cookies that were previously mentioned. CFI uses a finite static Control Flow Graph to restrict unintended Control Flows from those intended by the programmer. Despite being an effective method, it has its drawbacks. Firstly, its high overhead produced when building the Control Flow Graph and checking every jump of the program if it is legitimate using labels makes it impractical to be integrated in production compilers. In addition, Control Flow Integrity builds its Control Flow Graph using source code which most of the times is not available in commercial software, as a result Control Flow Integrity can't protect from any Control Flow attacks.Despite Control Flow Integrity promising capabilities against Control Flow Hijacking, it is still prone to some attacks. For instance, code reuse attacks are able to bypass CFI using ROP [10] exploitation.

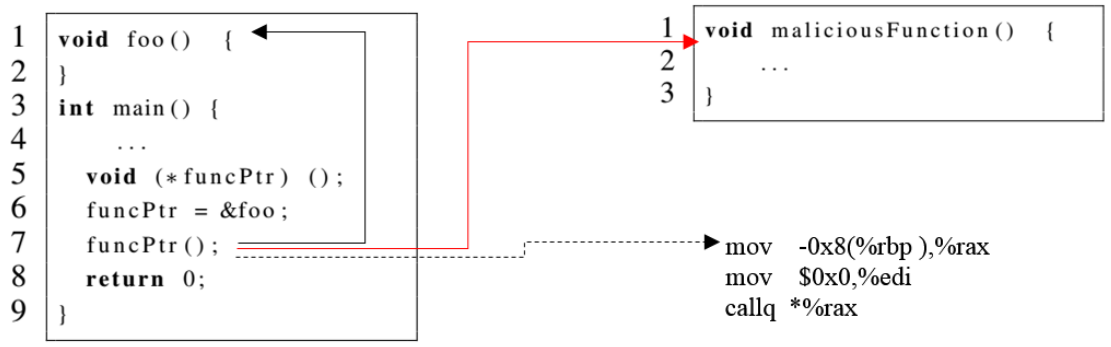


Figure 2.3: Control Flow Hijacking example. The above Figure illustrates a simplified example that shows that a program supposed to call the function "foo" but instead the attacker temper the control flow of the program by changing the control data "%rax" register and replace it with the "maliciousFunction" address .

2.7 Indirect Branches

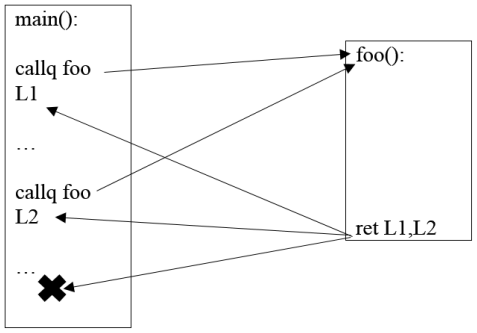
Indirect branches (Listing 2.7.1) are control program instructions that instead of having precomputed address for the next instruction that will get executed, they resolve that address at runtime. Having said that, attackers tamper control data, namely the addresses used for the calls thus achieve control flow hijacking, one of the currently most popular attacks in software exploitation. Below follow two examples of mechanisms provided by the programming languages C and C++ that utilize indirect branching.

2.7.1 Function Pointers

Programming language C uses function pointers (Listing 2.7.1) in order to achieve indirect branching. Basically, function pointers are used for calling functions but what differs from a direct branch is that when a function pointer is called, the address that holds at that specific moment of the call is the function address that is going to be called. The significance behind this, is the fact that the target address gets resolved at runtime rather at the compiling time like direct branches.

```
1  void foo()  {
2  }
3  int main() {
4
5    foo();
6    ...
7    foo();
8    ...
9    return 0;
10 }
```

a: C source code



b: Control Flow Integrity

Listing 2.6.1: Control Flow Integrity example. As shown by 2.6.1a, "foo" function is called in two different places in the "main" function. This created two lables (L1,L2) that permits foo to return. In other case "foo" cannot return. This can be seen by the final arrow that tries to return to a place but because it doesn't belong to a valid list of labels the return is failed.

2.7.2 Virtual Tables Pointers

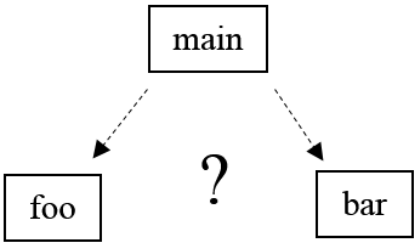
Programming language C++ uses virtual function tables called VTables for dynamic dispatching. As can be seen by Listing 2.7.2, Parent is a superclass and Boy and Girl are two subclasses. The input will determine if p1 will be instance of Boy or instance of Girl. As a result compiler doesn't know which of the two functions is going to be called. For this reason the correct function is going to be selected by the virtual function table which holds all the virtual functions during runtime.

2.8 Rust

Rust [20] is a systems programming language which statically enforces memory safety at compilation time in contrast with other programming languages which enforce memory safety at runtime. This happens for efficiency reasons. In addition, Rust uses a packet manager called Cargo to provide Foreign Function Interface(FFI). Specifically it provides efficient C bindings.

```
1  void foo()  {
2  }
3  void bar()  {
4  }
5  int main() {
6       ...
7    void (*funcPtr) ();
8    if input < 0
9      funcPtr = &foo;
10   else
11     funcPtr = &bar;
12   funcPtr();
13   foo();
14   return 0;
15 }
```

a: C source code



b: Indirect Branch Illustration

Listing 2.7.1: Indirect Branch example. As can be seen by 2.7.1a, the program uses both direct and indirect branches. Line 13 which includes the direct branch, during compilation time the target address is known so it is a fixed value in the executable's binary code. However Line 12 includes an indirect branch which its target address will be determined by the input during runtime. As a result the indirect branch has a variable target address which can be changed during runtime contrary to the direct branch which has a fixed value

2.9 Capstone

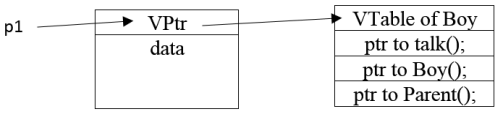
Capstone [19] is an open source multi-architecture disassembly framework which has been used as the core disassembler for the executables in the mbDisass.

2.10 Nucleus

Nucleus [4] is an open-source function detection tool for binaries which has been used in mbDisass to provide further information about the indirect branches, namely to split up the disassembly into functions and give us a more precise analysis. Nucleus detects functions at the Control Flow Graph - level, this means that it can detect functions regardless if the binary has been stripped.

```
1  class Parent{
2     public: virtual void talk(){}
3  };
4
5  class Boy: public Parent  {
6     public:void talk(){}
7  };
8  class Girl: public Parent  {
9     public:void talk(){}
10 };
11
12 int main(int argc, char *argv[]) {
13
14    Parent *p1;
15    input == true  ? p1 = new Boy() : p1 = new Girl();
16    p1->talk();
17    return 0;
18 }
```

a: C++ source code



b: VTable example

Listing 2.7.2: VTable example. 2.7.2a shows a C++ program which uses dynamic dispatching. As it can be seen by 2.7.2a it is not known whether p1 is an instance of Boy or an instance of Girl, this will be determined during runtime. Thus when "talk" function is called in line 16 the virtual table will find the correct offset of the targeted function and call it.

2.11 LLVM Compiler Infrastructure

The LLVM compiler infrastructure [13] provides modular compiler technologies. Its design gives the ability to add custom passes to the compiler flow that perform transformations, optimizations and build analysis results. What's important with the LLVM compiler Infrastructure is the fact that all the phases in compilation are represented using LLVM Intermediate Representation (IR).

2.12 LLVM Intermediate Representation

LLVM compiler infrastructure uses Intermediate Representation(IR) throughout all its compilation phases. The purpose of this, is not only to provide a human readable assembly language representation but also to make analysis and debug a lot easier.

2.13 SQLite3

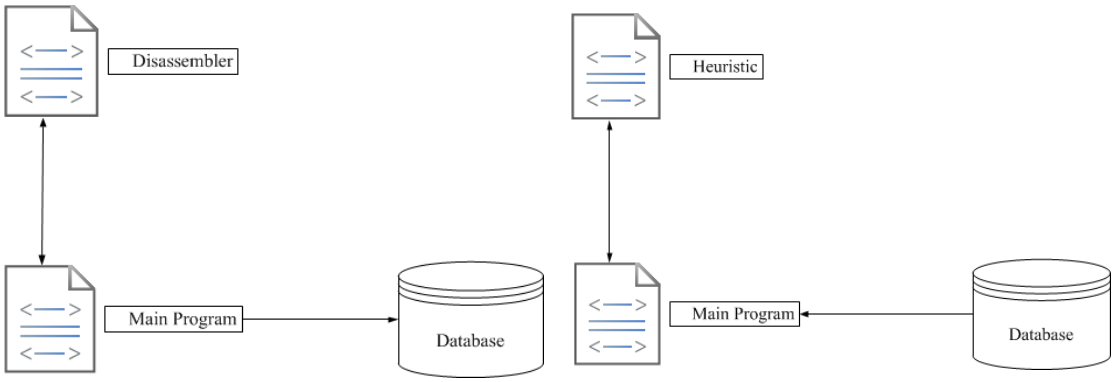
SQLite3 [16] is a public domain relational database management system written in C that provides applications, the capability for internal data storage.

Chapter 3

Architecture

mbDisass is static analysis tool for binaries that tries to identify every indirect branch contained in an executable/library and determine their origin programming language using heuristics rules. It is divided into two main sections, the Disassembler and the Heuristic. The Disassembler which loads the executable's binary and disassembles it in batches, finds the functions' entry points and combines them with the disassembly to create structures that hold the disassembled instructions separately based on the function they belong to.

Afterwards, these structures are saved in a persistent database to eliminate the need for repeating the section of Disassembler in future executions unless the binary changes. Finally, the Heuristic gets called. It loads the structures created by the previous section from the database, it finds the indirect branches and analyses them using heuristic rules to classify them in each of the three programming languages, C, C++ and Rust.



(a) Disassembler (b) Heuristic

Figure 3.1: mbDisass Architecture

3.1 Disassembler

The Disassembler section is divided into two phases:

1. Binary analysis and disassembly: Using conditions, the binary is divided and gets disassembled in batches.
2. Structures creation and database creation: Combining the functions' entry points with the executable's disassembly to create data structures that hold each function separately and then save them to a persistent database.

3.1.1 Binary analysis and disassembly

Accurate disassembly is crucial for static binary analysis. Through the various tests with mbDisass it showed that when the binary code that is going to be disassembled receives some kind of preprocessing, results in a more accurate disassembly. For this reason the objective of this step is to divide the binary into batches and then disassemble each batch separately. This approach helps disassembly overcome problems when different challenging cases arrive like extreme uses of padding.

3.1.2 Structures and Database creation

This step combines information about the executables' functions, namely entry points and sizes with the executable's disassembly to create structures which hold the information mentioned before. This happens because when the Heuristic phase begins and the heuristic rules start analysing the different functions, they isolate the area close to the actual indirect branch to recognize a specific programming language using a small number of instructions as some patterns need precision in order be successfully recognized by the correct heuristic rule. In addition, to avoid repeating the whole Disassembly section in future executions when having an already analysed executable, the structures are saved to a persistent database that can be retrieved at any time by the Heuristic to perform its analysis.

3.2 Heuristic

The analysis following the disassemble of the executable is divided into two phases:

1. Indirect Branches Extraction: Using static analysis to extract the indirect branches.
2. Indirect Branches Language Classifying: Using a set of heuristic rules to determine the origin programming language.

3.2.1 Indirect Branches Extraction

After retrieving the structures containing the functions from the previous step from the database, the indirect branches finding begins. The simple pattern that the indirect branches have at the binary level makes it easy to extract them by a single pass of the disassembly. When an instruction is a call or a jump to a register, then it is considered as an indirect branch by mbDisass.

3.2.2 Indirect Branches Language Classifying

Now that the indirect branches have been obtained from the disassembly, is time to try to classify them to any of the three languages C, C++ and Rust. We used several heuristic rules that each one serves one of the three languages mentioned above, but every rule follows the same approach. The heuristic rules use a number of instructions appearing above the indirect call to identify the origin programming language. When a rule gets enabled, the percentage of the language that was written for, increases. At the end, the language with the highest percentage is considered the corresponding language of the indirect branch.

Heuristics Rules The heuristics rules are basically used for classifying the indirect branches. Each rule was written to identify a single pattern produced by a single programming language. The reason that indirect branches classifying is not a standardised method is because there isn't a single pattern that can determine the producing programming language of an indirect branch.

```
1  void indirectFuncCallee() {
2
3  }
4
5  int main() {
6
7  void (*funcPointer) ();
8    funcPointer = &indirectFuncCallee;
9    funcPointer();
10   return 0;
11 }
```

a: C source code

```
1  4004d6 <indirectFuncCallee>:
2      ...
3
4
5  4004e0 <main>:
6
7      ...
8  mov    -0x8(%rbp),%rax
9  mov    $0x0,%edi
10 callq  *%rax
11     ...
```

b: Disassembly indirect branch

Listing 3.2.1: The above listing demonstrates the C pattern occurring when an indirect branch is created by a function call using a function pointer . As it can be seen when the function is called using a function pointer, a "mov" instruction(Listing 3.2.1b - Line 8) appears that moves the address of the target function to the calling register.

As shown in Listing 3.2.1b (Lines 8 - 10) there are three lines of disassembly produced by the C indirect branch. The first "mov" instruction represents the move of the callee address to the calling register. The second "mov" instruction represents the arguments taken by the callee function, in this case there are none. Finally the call instruction represents the indirect branch to the address , namely "foo" using the register "rax".

```
1  fn  indirect_func_callee(){
2  }
3
4  fn  indirect_func_caller(func_pointer: fn
       ()){
5     func_pointer();
6  }
7
8  fn main(){
9
10    indirect_func_caller(
          indirect_func_callee);
11
12 }
```

a: Rust source code

```
1  6db0 <...indirect_func_callee...>:
2       ...
3
4  6dc0 <...indirect_func_caller...>:
5       ...
6  callq  *%rdi
7       ...
8
9  6dd0 <...main...>:
10      ...
11 push   %rax
12 lea    -0x28(%rip),%rdi        # 6db0
13 callq  6dc0 <...indirect_func_caller...>
14      ...
```

b: Disassembly indirect branch

Listing 3.2.2: The above listing demonstrates the Rust pattern occurring when an indirect branch is created by a function call using a function pointer . As it can be seen, above the indirect branch(3.2.2b - Line 6) there isn't any "mov" instruction that is used for transferring the target function address to the calling register. This is due to the fact that the target address was already transferred to the calling register in the "main" function(3.2.2b - Line 12) because the target function was sent as argument to the function "indirect_func_caller".

As shown in Listing 3.2.2b - Line 6) there is only one line of disassembly, the actual call. This happens due to the fact that contrary to the C program, in the Rust program the function pointer gets "foo's address from main as an argument so there is no "mov" instructions above the actual call.

As a result of this example, we can conclude that despite the fact that in both situations a function pointer is used to call a function(foo), C and Rust have both similarities and differences in the patterns they produce.

Chapter 4

Implementation

mbDisass consists of two sections, the disassembler which is responsible for the preparation of the data structures containing the disassembly of the loaded ELF file and the Heuristic which is responsible for finding and classifying each indirect branch to a programming language. mbDisass was was built using python version 2.7 and uses only libraries provided by the language or open-source software.

4.1 Disassembler

The first step of our analysis starts by calling Nucleus [4]. As Nucleus is written in C++ with no python binding we used subprocess.process() to spawn a new process in order to execute Nucleus and also have access to its output. After Nucleus finishes, it returns the

functions' entry points and sizes which will be used later for splitting the disassembly of the ELF into functions.

Then mbDisass uses the open-source python library pyelftools [5] to load the ".text" section and its starting address of the ELF file in hexadecimal notation. The binary code of the ".text" section is loaded in lines of sixteen bytes each. ‘

Prior calling the Capstone disassembler, the hexadecimal representation of the ".text" section is being processed using regular expressions and escape character removal to transform it in the representation that Capstone can process, for example Listing 4.1 shows that '31ed' transforms into '\x31\xed'). As mentioned before, disassembly is an

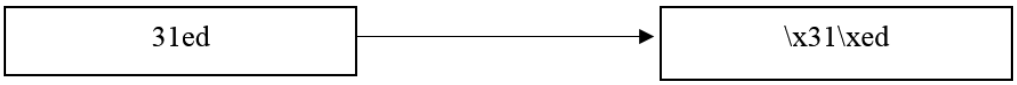


Figure 4.1: Hexadecimal preprocessing before Capstone example. When pyelftools load an executable, the binary is represented like the first box. In order for Capstone to disassemble the binary, it has to be in the format shown in the second box. For this reason the binary showed in the first box gets formatted to the binary showed in the second box.

unsolved problem, thus despite Capstone's outstanding capabilities, sometimes it seems to fail to return a precise disassembly when given a whole executable binary at once. As a workaround to this problem, we analyse the executable's binary to detect padding and disassemble it in batches, this results to better disassembly and consequently to a more accurate indirect branch detection. When splitting the binary code in the wrong places results in an inaccurate disassembly, for this reason we have added some conditions to ensure the regular flow of the program. Nevertheless these conditions are very time-consuming which affect the final runtime performance of the Disassembler.

For example Listing 4.1 shows some lines of binary code taken from Mozilla's Firefox library libmozavcodec.so. It is noticeable that Line 3 contains excessive padding which results in Capstone failing to disassemble anything after Line 3. The conditions mentioned before take care of these kind of situations as they split the binary code in batches and also verify that the batches won't make the Capstone to fail to generate the corresponding disassembly. As a result this is the most time-consuming step of the Disassembler

Capstone is adjusted to produce AT&T representation of the disassembly as the heuristic rules that will be discussed later are written with AT&T representation in mind. Moreover, we used the Capstone's settings that produce the disassembly in 64-bit code for the

```
1                ...
2     0d7dae13 00e9dbf9 ffff660f 1f840000
3     00000000 00000000 00000000 00000000
4     5581faff ffff0fb8 b7b1bbbe 4889e50f
5                ...
```

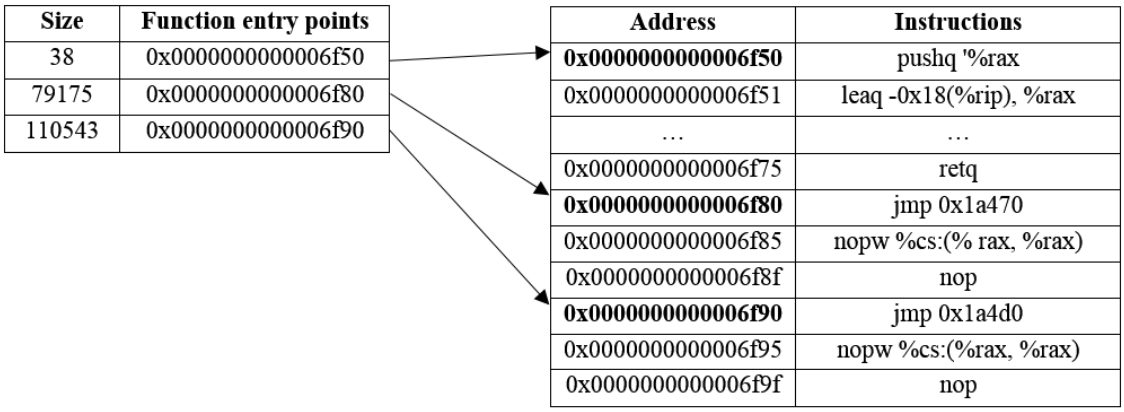
Listing 4.1: Binary from Mozilla's Firefox libmozavcodec.so in hexadecimal notation example. Line 3 has excessive padding which makes Capstone to stop disassembling anything below Line 3. Even though this is an example, its very common in commercial software

x86 architecture.

As Nucleus and Capstone return the hexadecimal addresses in different representation (for example, Figure 4.2 shows that Nucleus returns 0x00000000004003e0, in contrast to Capstone that returns 0x4003e0) we used a string formatting pre-processing to bring Capstones' addresses in Nucleus's representation.

Figure 4.2: Address Formatting example. Capstone returns instructions' addresses in the format showed in the first box, on the other hand, Nucleus returns functions' entry points in the format showed in the second box. For consistency reasons Capstones' addresses are transformed in the format represented in the seconds box.

Afterwards, we combine both the functions' entry points and sizes with the disassembly of the ".text" section of the ELF file to create data structures that hold every function of disassembly separately with their corresponding entry points and sizes. We used the entry points addresses and the functions' sizes to determine the last address of each function. By iterating simultaneously the functions' entry points and the disassembled instructions, we match each function entry point with the the instructions that correspond to its function boundaries. One challenge we face in this phase is that Nucleus in rare cases returns a size not reflecting the real size of the function, as a result, that function matches more instructions than what it actually has. To solve this problem, we check if the current instruction that is going to be matched with a function has smaller address than the next entry point in line. For example Figure 4.3, we have the entry points A=0x6f80 with function size 79175 bytes and the following entry point is B=0x6f90, the current inspecting instruction is I=0x6f90, what is going to happen here, before assigning I to A it checks if



Size	Function entry points
38	0x0000000000006f50
79175	0x0000000000006f80
110543	0x0000000000006f90

Address	Instructions
0x0000000000006f50	pushq '%rax
0x0000000000006f51	leaq -0x18(%rip), %rax
…	…
0x0000000000006f75	retq
0x0000000000006f80	jmp 0x1a470
0x0000000000006f85	nopw %cs:(% rax, %rax)
0x0000000000006f8f	nop
0x0000000000006f90	jmp 0x1a4d0
0x0000000000006f95	nopw %cs:(%rax, %rax)
0x0000000000006f9f	nop

Figure 4.3: Function Structure creation example. The example presented above shows how the function structures are created, namely the instructions get matched to their corresponding function using the their addresses. By combining the entry points and the sizes of each function we get their boundaries

I is higher or equal to B entry point and then it proceeds, as we can see for this example I isn't going to be assigned to A (actual example from a rust executable) but to B. Another challenge we face in this phase is that Nucleus in rare cases misses some function entry points. In these situations, we add all the instructions into one data structure to continue to the indirect branch detection by the Heuristic but with no function information.

mbDisass can be executed in two modes, one that does the above procedure only for the input file and another mode that also includes the dynamic dependencies of the input file. As a result the Heuristic finds both the indirect branches of the executable itself and also the indirect branches of the dynamic dependencies.

As mentioned before, the phase of the Disassembler creates enough overhead that the need for making this section redundant, arise. In order to achieve that, after the structures are created, they get saved in a database using SQLite3 [16]. As a result, unless an executable has changed, the Disassembler section needs to run once for each executable/library, then the Heuristic section can run at any time. An example representing how the data is saved in the database can be seen from the Table 4.1 . Each row represents an object containing a function with its instructions. The first column represents the name of binary file that contains the current function, the second column represents the entry point of the function, namely the beginning address of the current function, the third column represents the size of the current function which combined with the entry point said before is used to locate each function's boundaries. Lastly, the fourth column represents a serialized object of the function corresponding to the current row. The Python library cPickle [17] was used for serializing the objects. By serializing the objects, they get transformed in a format that

executable/library	Function Entry Point	Function Size	Function
Cexample.o	0x00000000004003e0	42	<Serialized Object 1>
Cexample.o	0x0000000000400410	50	<Serialized Object 2>
...	...	...	...
libc.so.6	0x000000000001f8b0	130	<Serialized Object 3>
libc.so.6	0x000000000001f932	74	<Serialized Object 4>
...	...	...	...
ld-linux-x86-64.so.2	0x0000000000000ac0	331	<Serialized Object 5>
ld-linux-x86-64.so.2	0x0000000000000c0b	30	<Serialized Object N>
...	...	...	...

Table 4.1: Database Function table example. Each row represents a function. The first column holds the name of the file the function is contained. The second column holds the entry point of the function. The third column holds the size of the function and finally the fourth column holds the serialized object of the function.

can be stored and can be accessed at any time by deserializing them.

4.2 Heuristic

Now that the data structures containing the disassembly of the ELF file divided into functions are saved in the database, the Heuristic gets called to find the indirect branches inside the disassembly and try to classify them to their corresponding programming language between C, C++ and Rust.

Firstly mbDisass accesses the database to retrieve all functions of the executable that is going to be analysed including the one structure containing all the instructions that their corresponding function wasn't found. By deserializing the serialized objects of functions, recreates the structures and prepares them for the analysis. Afterwards, the Heuristic tries to identify and classify the indirect branches using regular expressions [18] based on the AT&T notation, as a result every future expand to the Heuristic should be written with the AT&T notation in mind.

The Heuristic checks every instruction in the data structures containing the disassembly of the ELF file. If an instruction contains the pattern "callq *..." or "jmpq *..." then both the instruction and the function containing it, get classified as an indirect branch and

get saved for further analysing later. For example, if an instruction is "callq *%rax" then it is classified as an indirect branch.

Following the inspection for indirect branches, the functions that got classified as indirect branches, continue to the next phase that the Heuristic classifies them to their corresponding programming language. After observing different instruction patterns among the programming languages C, C++ and Rust regarding the indirect branches with hands on disassembly, we wrote some heuristic rules that try to determine the origin language when these patterns show up. Since there isn't any specific pattern for each language, the Heuristic sometimes fails to determine the programming language of some indirect branches.

The heuristic rules utilize the information extracted from the instructions appearing close to the actual indirect call in order to recognize any pattern that was mentioned above. The register used for the indirect branch, as other registers used for moving the address of the target function or any arguments used by the target function, are extracted from the instructions using regular expressions.

When a pattern is recognized, a counter of the language that was written for, gets increased by one. Every heuristic rule has its own weight, specifically when a rule gets activated, before its added to it's corresponding programming language counter, it gets multiplied with a number between 0.5 and 1.5 called weight. The reason behind this decision is because some of the patterns show up in more than one programming language. As a result the rule that gets activated for many languages won't determine the outcome by it self if it has a low weight. However if a pattern shows up specifically for one programming language, for example a function calling through a VTable, as this happens explicitly in C++, the rule corresponding to this pattern has a higher weight in order to determine the outcome.

Finally, the indirect branch gets assigned to the language with the highest percentage. This procedure is repeated for all the indirect branches found in the disassembly.

4.3 Heuristic Rules

As mentioned before, there isn't any specific pattern produced by the indirect branches that can be used to determine every programming language. Because of this, mbDisass

uses a number of heuristic rules that try to recognize the patterns that are produced by a single programming language. Currently, mbDisass searches for patterns produced by the programming languages C, C++ and Rust.

4.3.1 C++ Heuristic Rules

Heuristic Rule 1 The most distinct pattern that show up when an indirect branch gets enabled in C++ is the call of a virtual table function. As shown in the Listing 4.3.1, when a function is going to be called through a virtual table, we expect to notice one "mov" instruction which transfers the address to the calling register. In addition below the said "mov" instruction we expect an "add" instruction which adds to the address the offset for the targeted function. As a result this heuristic rule gets enabled when the Lines 11,13 of Listing 4.3.1b show up.

```
1  class Parent{
2     public: virtual void talk(){}
3     public: virtual void laugh(){}
4  };
5
6  class Child: public Parent  {
7     public:void talk(){}
8     public:void laugh(){}
9  };
10
11 int main(int argc, char *argv[]) {
12
13    Parent *p1;
14    p1 = new Child();
15    p1->talk();
16    p1->laugh();
17    return 0;
18 }
```

a: C++ source code

```
4006c6 <main>:                          1
                                        2
    ...                                 3
mov    -0x18(%rbp),%rax                 4
mov    (%rax),%rax                      5
mov    (%rax),%rax                      6
mov    -0x18(%rbp),%rdx                 7
mov    %rdx,%rdi                        8
callq  *%rax                            9
    ...                                 10
mov    -0x18(%rbp),%rax                 11
mov    (%rax),%rax                      12
add    $0x8,%rax                        13
mov    (%rax),%rax                      14
mov    -0x18(%rbp),%rdx                 15
mov    %rdx,%rdi                        16
callq  *%rax                            17
    ...                                 18
```

b: Disassembly indirect branch

Listing 4.3.1: Heuristic Rules 1,2 C++ example. The above listing demonstrates the C++ pattern occurring when an indirect branch is created by a virtual function call. As it can be seen, when a virtual function is called, a "mov" instruction(Listing 4.3.1b - Line 11) appears that moves the address of the virtual table to the calling register. Afterwards an "add" instruction is expected to appear(Listing 4.3.1b - Line 13) that adds the appropriate offset to get the targeted function address from the virtual table.

Heuristic Rule 2 Despite the fact that indirect branches created by virtual function calls are distinct because of the reasons mentioned in the previous C++ Heuristic rule, this is not always the case. As it can be seen from Listing 4.3.1, there are two virtual function calls, one to the function "talk"(Listing 4.3.1a- Line 15) and one to the function "laugh" (Listing 4.3.1a- Line 16) but only the "laugh" has the "add" instruction which is used for adding the offset in order to get the targeted function address. This happens because talk happens to appear first in the "Child" class, as a result the transferred virtual function address already points to the "talk" function and so no any other actions is needed. To overcome this issue, this rule is created which tries to identify the "mov" instructions pattern seen on the Listing 4.3.1b- Lines 4-6 without the need for the "add" instruction.

4.3.2 C Heuristic Rules

Heuristic Rule 3 By observing the disassembly patterns produced by C function pointers, it is easily noticeable that when an indirect pointer is used for calling a function then a "mov" instruction shows up that transfers the function address to the calling register and then at least one "mov" instruction which is used to transfer the arguments of the callee function to their corresponding registers. When the function gets called indirectly, and it takes more than one arguments, then the arguments transference show up as "mov" instructions, between the "mov" instruction of the target function address to the calling register and the actual "call" instruction. Even if the function doesn't take any arguments, there is still one "mov" instruction other than the one that transfers the address which transfers a "0x0" Listing 4.3.2b - Line 9.

Heuristic Rule 4 This rule is meant for detecting patterns that occur in case a function pointer is used to call a function that takes arguments. As mention in the Heuristic Rule 3, when a function takes arguments, "mov" instructions show up between the "mov" instruction of the target function address to the calling register and the actual "call" instruction (Listing 4.3.3b - Lines 8,9).

```
1  void indirectFuncCallee()  {
2  }
3
4  int main() {
5
6  void (*funcPointer) ();
7    funcPointer = &indirectFuncCallee;
8    funcPointer();
9    return 0;
10
11 }
```

a: C source code

```
1  4004d6 <indirectFuncCallee>:
2      ...
3
4
5  4004e0 <main>:
6
7      ...
8  mov    -0x8(%rbp),%rax
9  mov    $0x0,%edi
10 callq  *%rax
11     ...
```

b: Disassembly indirect branch

Listing 4.3.2: Heuristic Rule 3 C example.The above listing demonstrates the C pattern occurring when an indirect branch is created by a function call using a function pointer. As it can be seen when a function is called using a function pointer, a "mov" instruction(Listing 4.3.2b - Line 8) appears that moves the address of the target function to the calling register.

```
1  void indirectFuncCallee(int a,int b)  {
2  }
3
4  int main() {
5    void (*funcPointer) (int, int);
6    funcPointer = &indirectFuncCallee;
7    funcPointer(5,2);
8    return 0;
9
10
11 }
```

a: C source code

```
1  4004d6 <indirectFuncCallee>:
2      ...
3
4
5  4004e3 <main>:
6      ...
7  mov    -0x8(%rbp),%rax
8  mov    $0x2,%esi
9  mov    $0x5,%edi
10 callq  *%rax
11     ...
```

b: Disassembly indirect branch

Listing 4.3.3: Heuristic Rule 4 C example. The above listing demonstrates the C pattern occurring when an indirect branch is created when calling a function that takes arguments, indirectly . As it can be seen when function is called using a function pointer, a "mov" instruction(Listing 4.3.3a - Line 7) appears that moves the address of the target function to the calling register. In addition, two "mov" instructions appear (Listing 4.3.3b - Lines 8, 9) that are used for transferring the arguments that the callee function takes, into the registers.

4.3.3 Rust Heuristic Rules

Heuristic Rule 5 Rust also provides function pointers as C and C++, however it is observed that when a function pointer is declared in a function and assign a function address to it then when compiled it is turned into a direct branch by the compiler (Listing 4.3.4b - Line 9). Nonetheless as it can be seen from the Listing 4.3.5b - Line 6, function pointers in Rust can still produce indirect branches. When the address of a function gets sent as an argument to another function and the pointer holding the address is used to call that function then an indirect branch is produced. In this specific situation, unless the actual call happens immediately after the function entry, then no "mov" instruction is expected above the actual "call" instruction (Listing 4.3.5) in order for this heuristic to get enabled.

```
1  fn  indirect_func_callee(){
2
3  }
4
5  fn main(){
6
7    let func_pointer;
8    func_pointer = indirect_func_callee;
9    func_pointer();
10 }
```

a: Rust source code

```
1  6db0 <indirect_func_callee>:
2       ...
3
4
5  6dc0 <...main...>:
6       ...
7
8
9  callq  6db0 <...indirect_func_callee...>
10      ...
```

b: Disassembly indirect branch

Listing 4.3.4: Rust Function Pointer example. The above listing demonstrates that Rust function pointers written with the conventional manner don't create an indirect branch in contrast with other programming languages like C and C++. The reason behind this is the fact that Rust compiler replaces the indirect branch with a direct Branch (Listing 4.3.4b - Line 9).

```
1  fn  indirect_func_callee(){
2  }
3
4  fn  indirect_func_caller(func_pointer: fn
       ()){
5     func_pointer();
6  }
7
8  fn main(){
9
10    indirect_func_caller(
          indirect_func_callee);
11
12 }
```

a: Rust source code

```
1  6db0 <...indirect_func_callee...>:
2       ...
3
4  6dc0 <...indirect_func_caller...>:
5       ...
6  callq  *%rdi
7       ...
8
9  6dd0 <...main...>:
10      ...
11 push   %rax
12 lea    -0x28(%rip),%rdi        # 6db0
13 callq  6dc0 <...indirect_func_caller...>
14      ...
```

b: Disassembly indirect branch

Listing 4.3.5: Heuristic Rule 5 Rust example.The above listing demonstrates the Rust pattern occurring when an indirect branch is created by a function call using a function pointer . As it can be seen, above the indirect branch (4.3.5b - Line 6) there isn't any "mov" instruction that is used for transferring the target function to the calling register. This is due to the fact that the target address was already transferred to the calling register in the "main" function at Listing 4.3.5b - Line 12 because the target function was sent as an argument to the function "indirect_func_caller".

Heuristic Rule 6 When a function which takes arguments, gets called indirectly, then two things happen. Firstly, a "mov" instruction shows up above the actual call (Listing 4.3.6b - Line 10). In addition above the transference of the target function address, a number of "mov" instructions show up (Listing 4.3.6b - Line 9) that are used for the transference of the arguments. When these two things happen, then this heuristic rule gets enabled.

```
1  fn  indirect_func_callee(number : i32){
2
3  }
4
5  fn  indirect_func_caller(func_pointer: fn
       (i32)){
6     func_pointer(5);
7  }
8
9
10
11 fn main(){
12
13
14    indirect_func_caller(
          indirect_func_callee);
15
16 }
```

a: Rust source code

```
1  6db0 <...indirect_func_callee...>:
2       ...
3
4  6dc0 <...indirect_func_caller...>:
5       ...
6  mov     $0x5,%eax
7       ...
8  mov     %rdi,(%rsp)
9  mov     %eax,%edi
10 mov     (%rsp),%rcx
11 callq   *%rcx
12      ...
13
14 6de0 <main>:
15      ...
16 lea     -0x38(%rip),%rdi  # 6db0
17 callq   6dc0 <...indirect_func_caller...>
18      ...
```

b: Disassembly indirect branch

Listing 4.3.6: Heuristic Rule 6 Rust example.The above listing demonstrates the Rust pattern occurring when an indirect branch is created when calling a function that takes arguments, indirectly . As it can be seen when function is called using a function pointer, a "mov" instruction(Listing 4.3.6b - Line 10) appears that moves the address of the target function to the calling register. In addition, one "mov" instruction appear (Listing 4.3.6b - Lines 9) that is used for transferring the argument that the callee function takes, into the register.

Heuristic Rule 7 This heuristic rule searches for the pattern that Rust traits produce. As shown by Listing 4.3.7 - Line 15, there is a "mov" instruction that transfers the target function address to the calling register. In addition despite the fact that the callee function takes arguments, the "mov" instruction that transfers the argument to the register appears above from the actual call but it's not produce by the actual call as in the case of the Rust function pointer.

```
1  struct Dog { func_ptr : fn(i32) }
2  trait Animal {
3      fn new(func_ptr : fn(i32)) -> Self;
4  }
5  impl Animal for Dog {
6    fn new(func_ptr : fn(i32)) -> Dog {
7          Dog { func_ptr :func_ptr  }
8      }
9  }
10 fn indirect_func_callee(number:i32){
11 }
12
13 fn main() {
14     let  dog: Dog = Animal::new(
           indirect_func_callee);
15     (dog.func_ptr)(5);
16 }
```

a: Rust source code

```
1  6db0 <...Dog...Animal...new...>:
2         ...
3
4
5
6
7
8  6dc0 <...indirect_func_callee...>:
9         ...
10
11 6dd0 <...main...>:
12        ...
13   mov     $0x5,%edi
14        ...
15   mov     (%rsp),%rax
16   callq   *%rax
17        ...
```

b: Disassembly indirect branch

Listing 4.3.7: Heuristic Rule 7 Rust example.The above listing demonstrates the Rust pattern occurring when an indirect branch is created by a function call through a trait. As it can be seen, when a function is called using a trait function pointer, a "mov" instruction(Listing 4.3.7b - Line 15) appears that moves the address of the target function to the calling register. In addition since the callee function takes an argument, a "mov" instruction shows in Listing 4.3.7b - Line 13 but it isn't created explicitly by the actual call instruction. Because of this, the "mov" instructions are not expected to appear immediately above the call instruction but could appear anywhere above it.

4.3.4 Previous Knowledge Rule

Heuristic Rule 8 Usually, some indirect branches can not be identified as they exhibit patterns that have not yet implemented as heuristic rules. As a result, these indirect branches remain unidentified during the analysis. To deal with this incident, the Rule 8 was written. This rule checks if the indirect branch that is being currently analysed, belongs to a function which was already classified from an indirect branch analysed before. If this is true then this rule gets enabled and increments the counter of the corresponding programming language.

4.3.5 Build-In Heuristic Rules

Heuristic Rule 9 During disassembly analysis it was noticed that some of the indirect branches were produced by the compiler. In order for these indirect branches not to interfere with the analysis of the others produced by the actual source code, we created a rule that classifies those produced by the compiler as Build In indirect branches.

```
1  400410 <deregister_tm_clones>:
2          ...
3    test   %rax,%rax
4          ...
5    jmpq   *%rax
6          ...
7  400450 <register_tm_clones>:
8          ...
9    test   %rax,%rax
10         ...
11   jmpq   *%rax
12         ...
13 4004b0 <frame_dummy>:
14         ...
15   test   %rax,%rax
16         ...
17   callq  *%rax
18         ...
```

Listing 4.3.9: Heuristic Rule 9 Rust example. The above listing illustrates three indirect branches produced by the compiler. As it can be seen above the actual callq/jmpq instruction, a "test" instruction is expected to appear including the register that holds the address of the target function that will be called indirectly.

4.4 LLVM IR - Bitcode Analysis

During the implementation of the Heuristic for detecting indirect branches on binary level, we also implemented a Heuristic for detecting indirect branches on the Bitcode level (Intermediate Representation of LLVM). After some tests it was noticeable that the intermediate representation of LLVM doesn't carry as much information as the disassembly regarding the indirect branches. As shown in the Listing 4.4.1, both the IR and the disassembly come from the same source, but with the IR case, there isn't any information regarding the registers that will be used for the transference of either the arguments or the address of the target function address. In addition, LLVM intermediate representation analysis becomes even more difficult because in order to produce the intermediate representation of a program, the source code is required, which usually it is not publicly available when it comes to commercial software.

```
void indirectFuncCallee(int a)
{
}


int main() {

void (*funcPointer) (int);
  funcPointer = &
        indirectFuncCallee;
  funcPointer(5);
  return 0;
}
```

a: C source code

```
define void @indirectFuncCallee(i32 %a) #0 {
entry:
    ...
}

define i32 @main() #0 {
entry:
    ...
  %0 = load void (i32)*, void (i32)** %funcPointer,
         align 8
  call void %0(i32 5)
    ...
}
```

b: LLVM IR

```
4004d6 <indirectFuncCallee>:
...



4004e0 <main>:

...
mov     -0x8(%rbp),%rax
mov     $0x5,%edi
callq   *%rax

...
```

c: Disassembly

Listing 4.4.1: C program in both LLVM IR and Disassembly. Listing 4.4.1a shows a program in C containing an indirect branch. Listings 4.4.1b and 4.4.1c show the indirect branch from Listing 4.4.1a to demonstrate that the static analysis in the binary level gives more information in contrast to the LLVM IR level analysis. This is because binary level analysis produce register information which can be used to trace down the transference of the addresses and arguments.

Chapter 5

Evaluation

mbDisass is implemented on Linux for x86_64. In this chapter we evaluate mbDisass in the following four aspects:

1. Runtime performance of the Disassembler.
2. Runtime performance of the Heuristic.
3. Indirect Branch Finding.
4. Indirect Branch Classifying.

We evaluated mbDisass on Ubuntu 16.04 LTS running on an Intel Core i7-4710MQ CPU @ 2.50GHz and 8GB of Ram. All the the evaluations following were performed only on the ".text" section of the executables/ libraries.

5.1 Runtime performance of the Disassembler

The Disassembler phase loads the executable's binary code, splits it in batches, disassembles it, finds the function entry points and finally creates the function structures and stores them in a database. This phase needs to be executed once for each executable as long as there aren't any changes to the binary code.

executable/library	Binary Code Lines	Time(mm:ss.ms)
servo	3032888	33:32.98
firefox	10281	00:05.78
libc.so.6	86822	00:55.79
libstdc++.so.6	44027	00:26.99
libm.so.6	29041	00:16.34
ld-linux-x86-64.so.2	7641	00:04.64
libgcc_s.so.1	4069	00:02.34
libfreebl3.o	101	00:00.21
libfreeblpriv3.so	25261	00:13.81
liblgpllibs.so	4662	00:02.98
libmozavcodec.so	96984	01:01.80

Table 5.1: Runtime performance of the Disassembler. ***Binary Lines*** gives the number of lines containing 16 bytes of binary code in hexadecimal notation that the Disassembler split during disassembling. ***Time*** gives the time needed for loading the executable's binary, the disassembly and storing the function structures that ware created.

Table 5.1 shows how much time is needed for the Disassembler to finish for each executable/library. The more binary code lines an executable has, the more time is needed for the Disassembler phase to finish. This is not only due to the obvious reason that the bigger the executable the greater time is needed for disassembling it but also, as it was mentioned in a previous section (Section 4.2), before disassembling the binary code, it goes through some conditions in order to be split into batches to achieve high disassembly accuracy. Because of the fact that these conditions prevent Capstone disassembler from crashing, they do some time consuming checks. As a result in order to achieve both a high disassembly accuracy and a low runtime performance, the number of the times these checks are executed during the Disassembler phase are defined by the number of binary code lines included in each executable.

5.2 Runtime performance of the Heuristic

The Heuristic phase loads the functions structures from the database and analyses every instruction to find the indirect branches. If an instruction is an indirect branch then the whole function containing that instruction proceeds to the next step in order to be determined its origin programming language by the heuristic rules.

executable/library	Instructions	Indirect Branches Discovered	Time(mm:ss.ms)
servo	10562414	58425	04:00.33
firefox	41535	126	00:00.69
libc.so.6	334042	1714	00:07.29
libstdc++.so.6	169697	1819	00:04.79
libm.so.6	100300	10	00:01.30
ld-linux-x86-64.so.2	28661	124	00:00.55
libgcc_s.so.1	16554	25	00:00.22
libfreebl3.o	470	24	00:00.02
libfreeblpriv3.so	98374	341	00:01.81
liblgpllibs.so	19369	137	00:00.41
libmozavcodec.so	349114	1916	00:12.36

Table 5.2: Runtime performance of the Heuristic. ***Instructions*** gives the number of lines analysed for indirect branches. ***Indirect Branches Discovered*** gives the number of indirect branches found by the mbDisass during the Heuristic phase. Finally, ***Time*** gives the time needed for loading the function structures from the database created by the Disassembler and the time needed for running the heuristic rules on the functions classified as having an indirect branch, searching for patterns that could determine the origin programming language of the indirect branches.

The Runtime performance of the Heuristic as shown by the Table 5.2 is a result of the combination of the number of instructions contained in an executable and the number of indirect branches found. This happens because all the functions containing at least one indirect branch proceed to be analysed by the heuristic rules.

5.3 Indirect Branches Finding

The indirect branches contained in each executable were found by parsing the "objdump" output of each file using the same patterns used in mbDisass to identify instructions as indirect branches.

executable/library	Indirect Branches	Indirect Branches Found	Accuracy
servo	60516	58425	96.54%
firefox	126	126	100%
libc.so.6	1720	1714	99.65%
libstdc++.so.6	1846	1819	98.53%
libm.so.6	10	10	100%
ld-linux-x86-64.so.2	124	124	100%
libgcc_s.so.1	25	25	100%
libfreebl3.o	24	24	100%
libfreeblpriv3.so	342	341	99.70%
liblgpllibs.so	138	137	99.27%
libmozavcodec.so	1916	1916	100%

Table 5.3: Indirect Branches Finding. ***Indirect Branches*** gives the number of indirect branches of the analysed executable/library. ***Indirect Branches Found*** gives the number of indirect branches found by the mbDisass during the Heuristic phase. ***Accuracy*** gives the accuracy of the discovery of indirect branches based on how many indirect branches exist and how many were found by the Heuristic.

Table 5.3 shows that mbDisass manages to identify indirect branches with a high accuracy. However the accuracy could be increased if the disassembly was more accurate compared to the current disassembly. As it was mentioned before, the Disassembler phase has some conditions that split the binary code in batches in order to achieve a more accurate disassembly. Nevertheless these checks increase the runtime performance as they are very time consuming. Since there is a trade off between the runtime performance and the accuracy of the disassembly, the Disassembler is adjusted so that the disassembly has a high accuracy with a low overhead. This could change and have an even more accurate disassembly and consequently a more accurate indirect branch identification but with a massive overhead increase. As a result we chose to limit the numbers that the Disassembler splits the binary code into batches in order to achieve a lower runtime performance but a less accurate indirect branch identification.

5.4 Indirect Branches Classifying

Each function found to have at least one indirect branch, proceeds to this phase. Numerous heuristic rules try to identify known patterns which could determine the origin programming language of the indirect branch. The heuristic rules in this current evaluation can identify indirect branches patterns from three different programming languages C,C++ and Rust.

executable/library	Unidentified	Build-In	C	Rust	C++	Total
servo	808	10	4203	43349	10055	58425
firefox	29	4	4	40	49	126
libc.so.6	136	1	224	610	743	1714
libstdc++.so.6	51	0	133	1202	433	1819
libm.so.6	0	0	0	1	9	10
ld-linux-x86-64.so.2	9	1	6	59	49	124
libgcc_s.so.1	0	0	0	1	24	25
libfreebl3.o	0	0	0	2	22	24
libfreeblpriv3.so	8	0	22	154	157	341
liblgpllibs.so	11	0	3	86	37	137
libmozavcodec.so	2	3	1589	198	124	1916

Table 5.4: Indirect Branches Classifying. ***Unidentified*** gives the number of indirect branches that were classified as Unidentified, namely those that no heuristic rule got enabled during the analysis. ***Build-In*** gives the number of indirect branches that were classified as Build-In, namely those that have been added by the compiler. ***C*** gives the number of indirect branches that were classified as C. ***Rust*** gives the number of indirect branches that were classified as Rust. ***C++*** gives the number of indirect branches that were classified as C++.

Table 5.4 shows how the heuristic rules classified each indirect branch for each executable/library. As there isn't any way to know how many of them were classified correctly we show below a ground truth example.

5.4.1 Ground Truth

In order to obtain the ground truth about the mbDisass indirect branch classification capabilities we wrote small programs containing indirect branches from C++, Rust and C. Below follows an example of a mixed binary containing indirect branches from both C and Rust.

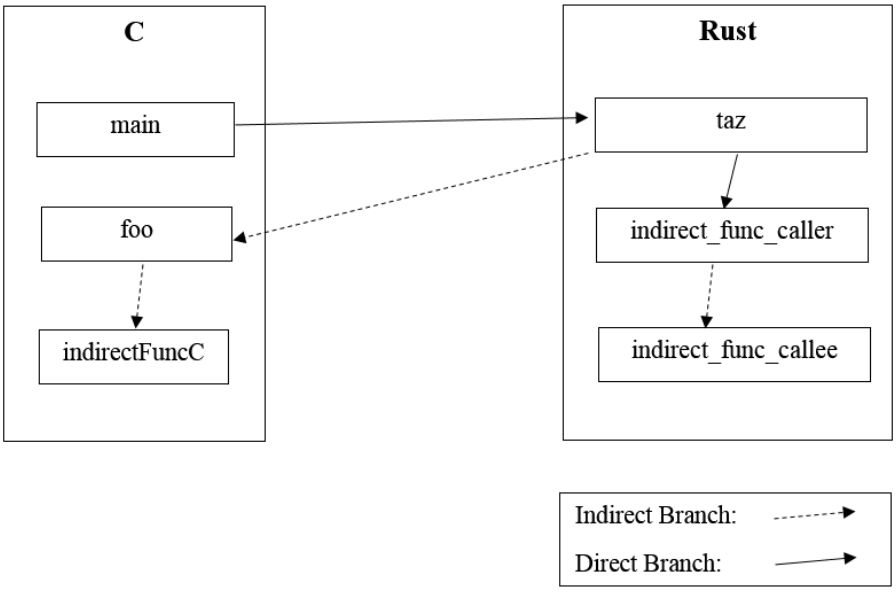


Figure 5.1: mixBinaryExample.o architecture. The above figure illustrates both the indirect and direct branches produced between the functions from different languages.

We wrote a small program named mixBinaryExample.o combining Rust and C, in order to obtain the ground truth. The mixBinaryExample.o's architecture can be seen from the Figure 5.1 which shows the direct and indirect branches. The mixBinaryExample.o's source code and disassembly is showed by the Listing 5.4.1. Finally, Table 5.5 shows mixBinaryExample.o's results when given to the mbDisass as the input.

```c
extern void taz();

void indirectFuncC(int a) {




}

void foo(){

  void (*funcPtrC) (int);
  funcPtrC = &indirectFuncC;
  funcPtrC(5);
  //Indirect Branch




}

int main() {

   taz(&foo);
   return 0;




}
```

```rust
#[no_mangle]
pub extern fn taz(func_ptr :fn()){




   indirect_func_caller(indirect_func_callee);
   func_ptr();    //Indirect Branch

}

fn indirect_func_caller(func_ptr: fn(x : i32,y :
     i32)) {
   func_ptr(5,2); //Indirect Branch




}

fn indirect_func_callee(x : i32,y : i32) {

}

extern{
   fn foo();




}
```

```
4004d6 <indirectFuncC>:
...



4004e0 <foo>:
...
mov    -0x8(%rbp),%rax
mov    $0x5,%edi
callq  *%rax
...

4004fe <main>:
...

400520 <taz>:
...
mov    %rdi,-0x8(%rbp)
...
callq  400550 <...
     indirect_func_caller...>
callq  *-0x8(%rbp)
...

400550 <...indirect_func_caller
     ...>:
...
mov    $0x5,%eax
mov    $0x2,%esi
...
mov    %rdi,-0x8(%rbp)
mov    %eax,%edi
callq  *-0x8(%rbp)
...
```

a: C source code b: Rust Source Code c: Disassembly

Listing 5.4.1: mixBinaryExample.o. The above listing illustrates a mixed binary example containing three indirect branches. It consists source code from both C (Listing 5.4.1a) and Rust (Listing 5.4.1b). The disassembly (Listing 5.4.1c) shows the indirect branches produced from each function.

5.4.2 False Negatives

Listing 5.4.1 and Figure 5.1 show that there are only two indirect branches coming from Rust and only one indirect branch coming from C. However mbDisass found two indirect branches coming from C and only one coming from Rust as can be seen by the Table 5.5. The reason behind this wrong classification is that the heuristic rules failed to classify the indirect branch contained in function "taz" as Rust because firstly, there aren't any arguments sent. This makes the pattern not as specific as it has to be in order to be classified as Rust. In addition the indirect branch is very close to the beginning of the function where the arguments' transference happens , as a result the core heuristic rule 3 which classifies indirect branches to Rust fails to get enabled.

Classification	mixBinaryExample.o
Unidentified	0
Build-In	4
C	2
Rust	1
C++	0
Total	7

Table 5.5: mbDisass's results of mixBinaryExample.o. Each row represents the number of indirect branches classified as the programming language specified by the first column.

5.4.3 Challenging Cases

In some cases its very difficult to distinguish the difference between the indirect branches because of numerous optimizations made by the compiler, several similarities between the different patterns or even unknown patterns that have not yet to be found. Some examples of challenging cases follow below:

- Tail calls: This is an example of the optimizations made by the compiler. The compiler instead of using the "call" instruction in order to call a function, it uses the "jmp" instruction. This optimization eliminates the need of the caller function to return to its caller, instead the callee function returns to its caller's caller function. The problem here is that because this is an optimization by the compiler, many indirect branches using the "jmp" instruction exhibit similar patterns across the three programming languages.
- Unknown Patterns: As it can be seen by the Table 5.4 there are many indirect branches that are classified as Unidentified. What happens in this cases is that no heuristic rule was activated. This is due to the fact that these indirect branches exhibit either a too generic pattern that cannot be implemented because it is similar across all the three programming languages or the pattern that they exhibit is unknown.

Chapter 6

Discussion - Future Work

6.1 Disassembler

As it was mentioned in previous sections, the Disassembler section of the program can produce a more accurate disassembly but this will result in a bigger overhead. The reason behind the significant overhead are different conditions that split the binary code into batches which prevent Capstone from failing to generate the corresponding disassembly.

An important modification that can be done in the future is to improve the Disassembler so that it can produce an accurate disassembly but without the current overhead. In order for this to happen, there is the need of better analysing the hexadecimal binary to reduce the different conditions that slow down the program.

6.2 Heuristic

As it can be noticed by the evaluation chapter. The mbDisass finds and correctly recognizes a significant number of indirect branches but still there is room for improvement. For that reason the more we understand the patterns exhibited by different indirect branches the stricter heuristic rules could be developed in order to get even better results.

Finally, since mbDisass offers extensibility, there is the possibility of developing heuristic rules that could classify indirect branches coming from programming languages other than C,C++ and Rust.

Chapter 7

Related Work

Analysing and defending control flow in modern software is a topic that draws great attention from the security research community as it is still considered an open problem.

7.1 Static Binary analysis

Before developing defences against control flow attacks, its essential first to understand a program's flow with the use of disassembly and static analysis tools. For example, Marx [15] presented by Pawlowski et al. is a class hierarchy reconstruction tool targeted at C++ binaries with high precision.

In addition mbDisass disassembly is based on linear disassembly which based on Andriesse et al. [3] research on x86/x64 Binaries disassembly accuracy, it was found that linear disassembly is generated with a high accuracy.

7.2 Control Flow Attacks and Defences

Many techniques have been introduced throughout the years that try to defend software from the control flow hijacking attacks. Now we will discuss some of the related work on

the defences proposed against control flow attacks.

Protections against VTable hijacking seem to draw a great attention. For example, VTrust [28] is a solution that focuses on protecting the virtual table calls. VTrust is a lightweight two layer defence implemented on the LLVM compiler. Its first layer focuses on validating that the runtime target virtual function has the same type with the one in the source code. The second layer ensures that the VTable pointers references a valid VTable. Similar work is VTint [27] against VTable pointers hijacking. Another similar proposal is VTPin [21] which preserves VTable pointers when their VTable objects gets freed in order to prevent their use in a "use-after-free" exploitation.

In addition to the defences mentioned above, there are some more generic like the core defence against control flow hijacking which is control flow integrity [1] which checks if the program follows its intended control flow.

Finally, the approach presented by Tice et al [26] is a great example of a CFI mechanism which ensures that Vtable pointers won't point to a malicious VTable with a little overhead. This solution was introduced as a gadget for both GCC and LLVM compilers.

Chapter 8

Conclusion

Binary static analysis is vital for understanding a program's control flow and developing defences against software exploitation. We have shown that mbDisass, our indirect branch analysis tool, finds and identifies a huge number of indirect branches among three programming languages Rust, C and C++. mbDisass can be easily extended to be able to identify indirect branches originated from programming language other than C, C++ and Rust.

Moreover mbDisass doesn't need source code as it conducts static analysis on binary level. Even though its Disassembler section adds a significant overhead, it is executed only once for each executable because its outcome gets stored in a database and can be accessed at any time to proceed with the Heuristic section.

It was showed that commercial software intermixes more than one programming languages. As a result, indirect branches originated from many programming languages coexist in the same executable. This could be a great hazard as indirect branches originated from languages that enforce memory safety at runtime, tamper control data coming from Rust, a language that doesn't enforce memory safety at runtime.

Bibliography

[1] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 340–353. ACM, 2005.

[2] S. Andersen and V. Abella. Data execution prevention. changes to functionality in microsoft windows xp service pack 2, part 3: Memory protection technologies, 2004.

[3] D. Andriesse, X. Chen, V. van der Veen, A. Slowinska, and H. Bos. An in-depth analysis of disassembly on full-scale x86/x64 binaries. In *USENIX Security Symposium*, pages 583–600, 2016.

[4] D. Andriesse, A. Slowinska, and H. Bos. Compiler-agnostic function detection in binaries. In *Security and Privacy (EuroS&P), 2017 IEEE European Symposium on*, pages 177–189. IEEE, 2017.

[5] E. Bendersky. Parsing ELF and DWARF in Python, 2018. `https://github.com/eliben/pyelftools`.

[6] N. Carlini and D. Wagner. Rop is still dangerous: Breaking modern defenses. In *USENIX Security Symposium*, pages 385–399, 2014.

[7] L. Clark. Inside a super fast CSS engine: Quantum CSS (aka Stylo), 2017. `https://hacks.mozilla.org/2017/08/inside-a-super-fast-css-engine-quantum-css-aka-stylo/`.

[8] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *USENIX Security Symposium*, volume 98, pages 63–78. San Antonio, TX, 1998.

[9] D. Dai Zovi. Practical return-oriented programming. *SOURCE Boston*, 2010.

[10] E. Göktaş, E. Athanasopoulos, H. Bos, and G. Portokalidis. Out Of Control: Overcoming Control-Flow Integrity. In *Proc. of IEEE S&P*, pages 575–589, 2014.

[11] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6–pp. IEEE, 2006.

[12] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *Performance Analysis of Systems & Software (ISPASS), 2010 IEEE International Symposium on*, pages 175–183. IEEE, 2010.

[13] LLVM. The LLVM Compiler Infrastructure, 2018. `https://llvm.org/`.

[14] A. One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):365, 1996.

[15] A. Pawlowski, M. Contag, V. van der Veen, C. Ouwehand, T. Holz, H. Bos, E. Athanasopoulos, and C. Giuffrida. Marx: Uncovering class hierarchies in c++ programs. In *Proceedings of the 24th Annual Symposium on Network and Distributed System Security (NDSSâĂŹ17)*, 2017.

[16] Python. DB-API 2.0 interface for SQLite databases, 2018. `https://docs.python.org/2/library/sqlite3.html`.

[17] Python. Python object serialization, 2018. `https://docs.python.org/2/library/pickle.html`.

[18] Python. Regular expression operations, 2018. `https://docs.python.org/2/library/re.html`.

[19] N. A. Quynh. Capstone: Next-gen disassembly framework. *Black Hat USA*, 2014.

[20] Rust. The Rust Programming Language, 2018. `https://doc.rust-lang.org/book/second-edition/index.html`.

[21] P. Sarbinowski, V. P. Kemerlis, C. Giuffrida, and E. Athanasopoulos. Vtpin: practical vtable hijacking protection for binaries. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, pages 448–459. ACM, 2016.

[22] F. J. Serna. Cve-2012-0769, the case of the perfect info leak. In *Blackhat Conference, Feb*, 2012.

[23] H. Shacham. The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 552–561. ACM, 2007.

[24] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, pages 298–307. ACM, 2004.

[25] L. Szekeres, M. Payer, L. T. Wei, and R. Sekar. Eternal war in memory. *IEEE Security & Privacy*, 12(3):45–53, 2014.

[26] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in gcc & llvm. In *USENIX Security Symposium*, pages 941–955, 2014.

[27] C. Zhang, C. Song, K. Z. Chen, Z. Chen, and D. Song. Vtint: Protecting virtual function tables' integrity. In *NDSS*, 2015.

[28] C. Zhang, D. Song, S. A. Carr, M. Payer, T. Li, Y. Ding, and C. Song. Vtrust: Regaining trust on virtual calls. In *NDSS*, 2016.