Control-Flow Integrity

MTP Report

Submitted in partial fulfillment of the requirements of the degree of
Master of Technology

by

Swapnil Raut
(153050086)

under the guidance of

Prof. R.K. Shyamasundar

Master of Technology Programme in
Computer Science and Engineering
Indian Institute of Technology Bombay

July, 2017

Acknowledgment

I have this opportunity to express my deep sense of gratitude and special thanks to my thesis guide **Prof. R. K. Shyamasundar** for his invaluable time, guidance and support.
I also extend my sincere thanks to Information Security Research and Development Center as well as the Department of Computer Science and Engineering for providing right ambience.

Swapnil Raut
M.Tech CSE
IIT Bombay

Abstract

Program execution must follow its execution path. Change in the path of the program execution can be used by an adversary to execute malicious code stored in memory. This malicious code execution leads to attacks such as buffer overflow, return-to-libc, return-oriented programming (ROP), etc. Control-Flow Integrity (CFI) is one of the mitigation techniques against such attacks. It enforces a program to follow the path of execution defined by Control-Flow Graph (CFG). CFI guarantees that program execution path can not be changed by an adversary and the program flow integrity is maintained. Our CFI implementation mainly includes generating regular expression of a program to validate path of execution and checking the integrity of return address stored on the stack. During runtime, complete path or history of execution is maintained, undefined change in the path of execution by an adversary is detected with the help of regular expression and by checking integrity of the return address stored on the stack.

Contents

List of Figures

Chapter 1

Introduction

Software programs have become part of our modern lives. Almost every event of our daily life is dependent on a hardware component that is controlled by a software. For example, kitchen appliances, building infrastructure like elevators and air-conditioning, telephone systems that connect our mobiles, satellites for TV and GPS, medical devices, etc. These systems involve some sort of micro-controller run by a software that allows us to operate and interact with these system. An error or bug in the software will have consequences. Therefore, rigorous testing is performed on such systems before they are deployed. Despite the best efforts, we come across events where software bugs cause disruptions in services they are responsible to control. Micro-controller based devices and their associated software is relatively easier to check and verify against presence of bugs due to its size. As we move to complex software like multi-user operating systems and the applications running on top of them; it becomes strenuously difficult to guarantee absence of bugs. Thus, there is always a possibility that the software does not run as intended – that is – it deviates from its intended functionality. Software bugs exist due to various reasons. We have listed the most important ones below:

1. poor programming practices,
2. insufficient testing or poor design of test cases,
3. complex applications
    (a) to address complexity and management, software is made modular
    (b) modules could be system libraries or packages
    (c) core application logic relies on modules that are written by others

Figure 1.1 shows possibilities of flow of control in a software that has several modular components: user written functions, or system libraries, or other applications that are present on the system.

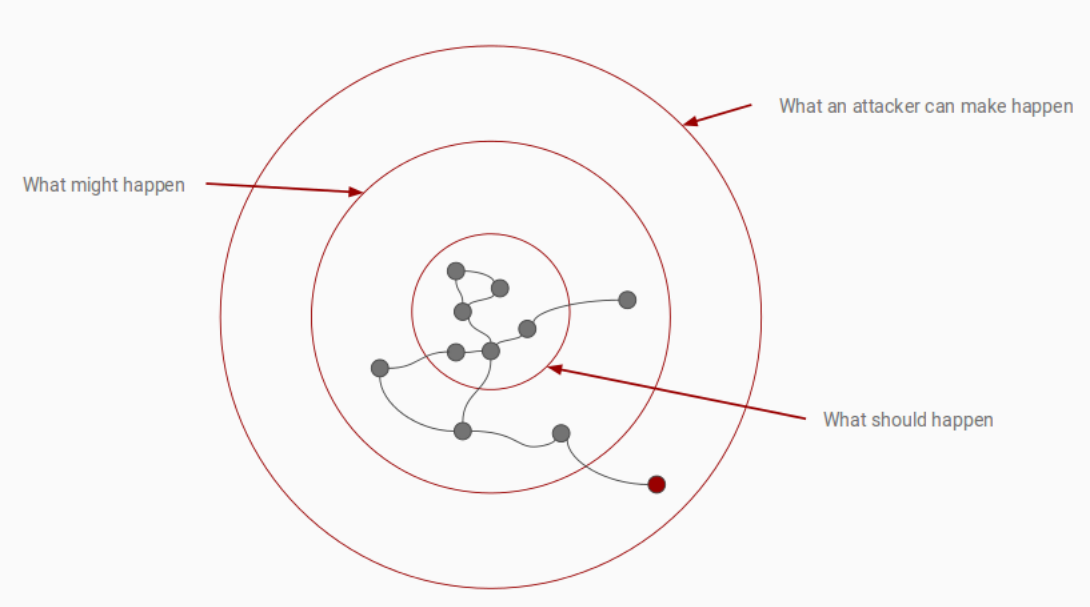

Figure 1.1: Control flow in a software (Credit: Stamatos)

Pushing software into exploitable state has become an *art* that many people master for fun or for profit [20]. The art involves predicting the kinds of mistakes engineers will make and quality assurance teams will miss. Therefore, it is of paramount importance that a software program should not be pushed into an exploitable state either by error or malicious intention.

In this work we have proposed and implemented a monitor function that ensures correct program execution path. Our approach is a type of CFI (control flow integrity) technique that requires function instrumentation. A regular expression is derived for a given program and the regular expression guides CFI checks. Our work is limited to C and C++ programs. Our approach can be extended to other programming environments.

Software written in C, C++ languages are likely to be unsafe due to various reasons. Programmers deliberately avoid the use of safety features such as strong typing and overflow detection for the sake of improved efficiency. A big class of software bugs or vulnerabilities that may lead execution of a software into exploitable state is buffer overflow. Skipping overflow detection may lead to buffer overflow attack. When a function is called, its activation record is stored on a stack. One

slot of activation record is reserved for buffer. An adversary can overflow data in the buffer, under its control, beyond its stated capacity. When data stored in buffer crosses buffer boundary, it causes buffer overflow. An adversary can use buffer overflow attack to change control-flow of the program. Several techniques and programming guidelines do exist to mitigate buffer overflow attacks. However, there are other ways to hijack control-flow of a program and push the software to exploitable state. A flow-based mitigation technique is used to check deviation of a program's execution path from its intended path. We shall introduce the reader to various types of return address corruption techniques in next chapter along with mitigation techniques that are proposed in literature. In Chapter 4, we shall introduce our approach based on regular expressions. We shall show how regular expressions succinctly represent all possible intended paths a program execution might take based on the call graph of functions present in the source ode of a given program. In Chapter 5, we show with the help of four different scenarios how our function instrumentation transparently act as a monitor for control flow integrity violations. We conclude with future works in Chapter 6.

Chapter 2

Background

Program is a sequence of instructions to perform a particular task. Every program has allocated memory which mainly consists of the code section, heap, stack, etc. Before program execution, it has to be stored in main memory for execution. The stack plays a very important role during program execution. The main purpose of the stack is to store activation records (stack frames) of functions. An activation record is divided into slots, and each slot is reserved for function arguments, return address, stack frame pointer, buffer, and callee saved registers. Figure 2.1 shows a stack on 32-bit Intel Architecture.

When a function is called, the activation record is pushed onto the stack. Stack frame pointer is used to point to the previous stack frame. Activation records are pushed on the stack in FILO (First In Last Out) order. When a function is called, it must return to the location from where it is called. Therefore, one slot of activation record is reserved for the return address. When a function is called, local variables, return address, and registers are temporarily stored in the stack. When a function returns, temporarily stored data on the stack is restored. The return address that was stored on the stack is used to return to the location from where the function was called. A part of stack memory is reserved for local variables. `%ebp` register is base pointer. All the function parameters and local variables are at a constant offset from base pointer `%ebp`. `%esp` is current stack pointer. When data is pushed on or popped off the stack, `%esp` changes.

2.1 Stack Corruption Techniques

In activation record, the stack has one slot reserved for the return address. When a function is called, the return address is stored on the stack. The return address

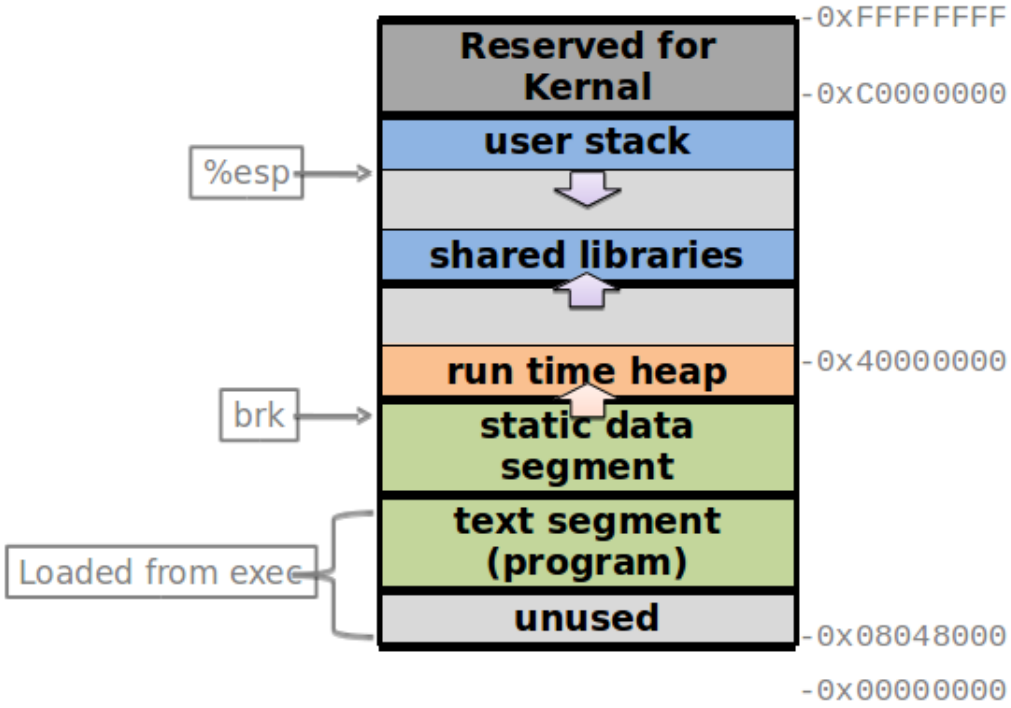

Figure 2.1: Linux process memory layout

points to the location where the function is supposed to return after its execution is over. An adversary can change the return address stored on the stack to the location where the malicious function is stored. Therefore return address is very important factor during program execution, and it is needed to be secured from an adversary.

2.1.1 Corrupting Callee-Saved Registers

To increase the efficiency of the program, the compiler tries to use registers instead of memory. Memory access takes more time compared to registers. However, when all registers are in use, and new process (program in execution) needs registers, some of the in-use registers are spilled on the stack. An adversary can take advantage of this to hijack control-flow of the program.

When one function (caller) calls another function (callee), all the registers which caller is using are temporarily stored on the stack. When a function returns, all the stored registers are restored. An adversary can corrupt these callee-saved registers. This can be a problem if callee uses restored registers for CFI check.

2.1.2 Corrupting System Call Return Address

The operating system has two modes of operation:

- Kernel-mode: All user programs execute in this mode.
- User-mode: All kernel programs execute in this mode.

. Whenever there is a system call, control moves from user-mode to kernel-mode. After execution of system call procedure, control returns to user-mode from kernel-mode. However, when control is shifting from user-mode to kernel-mode and back from kernel-mode to user-mode, there is a chance that adversary can take advantage of this control shift. The main reason behind this scope of the attack is that user-mode applications are completely separated from the kernel.

When a system call takes place, before shifting control to kernel mode, the return address is stored on the user-mode stack. While returning from kernel-mode to user-mode, kernel reads the return address from the user-mode stack. User-mode CFI instruments user-mode applications and not the kernel [2].

“A special instruction "sysenter" was introduced in x86 32-bit architecture to speedup the transition between kernel-mode and user-mode” [2]. The sysenter instruction execution does not save any state information. Therefore the return address is saved on the user-mode stack by Windows before executing this instruction. When a system call procedure execution is completed, kernel reads the return address from the user-mode stack. An adversary can overwrite the return address with the address of malicious code, and this leads to control-flow hijack and bypasses the CFI.

2.1.3 Buffer Overflow

When one function (caller) calls another function (callee) activation record (stack frame) is temporarily stored on the stack. The activation record has return address and buffer as shown in 2.2. An adversary can fill the data into the buffer more than buffer capacity. This causes buffer overflow. The return address stored on the stack can be overwritten with buffer overflow attack.

Figure 2.2 shows one slot is allocated to buffer in activation record of a function. Local variable are stored in buffer. Frame pointer is pointer to caller frame. One slot is reserved for return address.

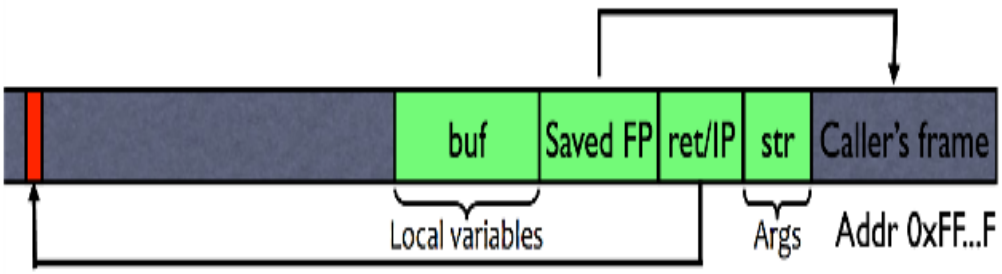

Figure 2.2: Buffer overflow

2.1.4 Return-to-libc

Return-to-libc is one of the attack techniques. It uses buffer overflow attack to execute code present in process executable memory. When the return address stored on the stack, an adversary triggers buffer overflow attack which replaces return address stored on the stack to address of subroutine present in the library. libc is standard C library. An adversary uses libc library for return-to-libc [18] attack. NX bit feature used to make some part of memory non-executable. NX bit feature is bypassed by this attack as it uses only executable code to trigger an attack.

2.1.5 Return-Oriented Programming

This is one of the security exploit techniques. An adversary tries to execute code in the presence of defenses such as $W \oplus X$. First, an adversary triggers buffer overflow attack to hijack control-flow of the program. An adversary executes chosen machine sequences present in memory [19]. Execution of chosen machine sequences allows an adversary to perform an arbitrary operation. Software must follow it's execution path. An adversary can hijack control-flow of the program and direct it to malicious code. Therefore we need to deal with attacks such as buffer overflow. The vulnerabilities in the software can be exploited by attackers to get memory access and trigger attacks such as buffer overflow, return-to-libc, ROP, etc. An adversary can use flaws in software to change control-flow of the program. Many C library functions are unsafe. C library function `strcpy` copies input data into buffer. `strcpy` function does not check the size of input. An adversary uses unsafe library functions to trigger buffer overflow attack. `strncpy` is introduced to overcome drawback of `strcpy`. `strncpy` copies input data of size n. However this is not concrete method to protect a program from buffer overflow attack.

2.2 Protecting Return Address

There are many mitigation techniques that have been proposed to protect return address on stack. Here we discuss such mitigation techniques.

2.2.1 StackGuard: canary

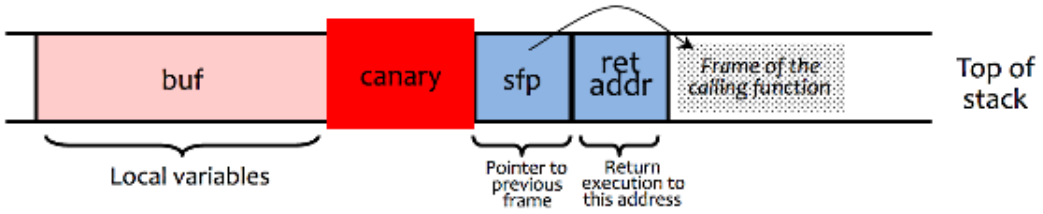

Figure 2.3: StackGuard: canary

In buffer overflow attack, an adversary tries to fill data into buffer more than buffer capacity. A buffer overflow occurs when data stored in buffer crosses buffer boundary. Stack canary [5] is one of the mitigation techniques which protects against buffer overflow attack. Canary is placed above buffer in the stack frame. It monitors buffer overflow. Whenever buffer overflow occurs, canary data gets corrupted or overwritten before return address gets overwritten. A canary is added just above buffer as shown in Figure 2.3. During buffer overflow, canary gets corrupted or overwritten before return address stored on the stack. Checking integrity of canary before function return helps to detect buffer overflow.

Drawback: A canary is embedded above buffer in activation record. An adversary can change the return address stored on the stack without changing canary. An adversary can store a pointer in the buffer. This pointer points to the return address stored on the stack. Using this pointer, an adversary can change the return address.

2.2.2 ASLR (Address Space Layout Randomization)

Address Space Layout Randomization (ASLR) is another mitigation technique. To change data stored on the stack, an adversary must know stack address. Randomization of memory layout reduces the chances of buffer overflow attack by making more difficult for an adversary to find stack address. Randomly arranged address space layout reduces the chances of buffer overflow attack by making more difficult for an adversary to find stack address.

Drawback: Over the years, powerful attacks have been invented that make an adversary more powerful. ASLR security technique randomizes only base pointer of the stack, heap, and library. The stack frame structure remains same.

2.2.3 PointGuard

This security technique protects function pointers stored in memory. It encrypts all the function pointers stored in memory. During program execution, a random key is generated. Whenever the pointer is stored in memory or retrieve from memory, the pointer is XORed with the key. When an adversary overwrites pointer stored in memory, after XORing, the pointer will point to a random location in memory.

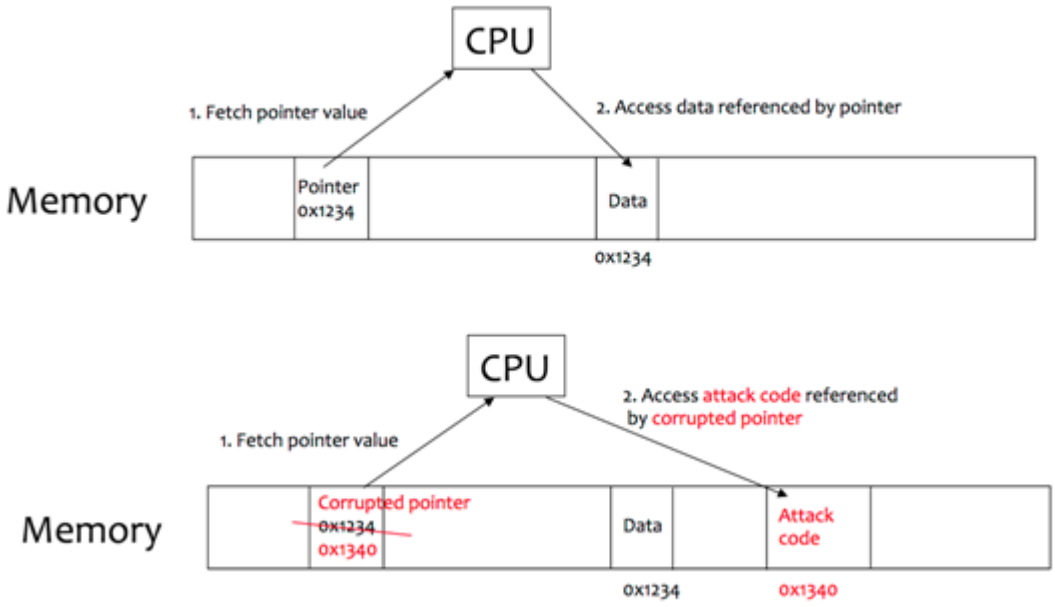

Figure 2.4: Normal point dereference [13]

Figure 2.4 shows normal pointer dereference. A function pointer is overwritten to point to attack code.

Figure 2.5 shows PointGuard dereference. A function pointer is overwritten to point to attack code. When pointer is fetched, it decrypts to random value.

Drawback: An adversary should not be able to change key. This technique must be fast. If unencrypted pointers are spilled on the stack, an adversary can change or corrupt them.

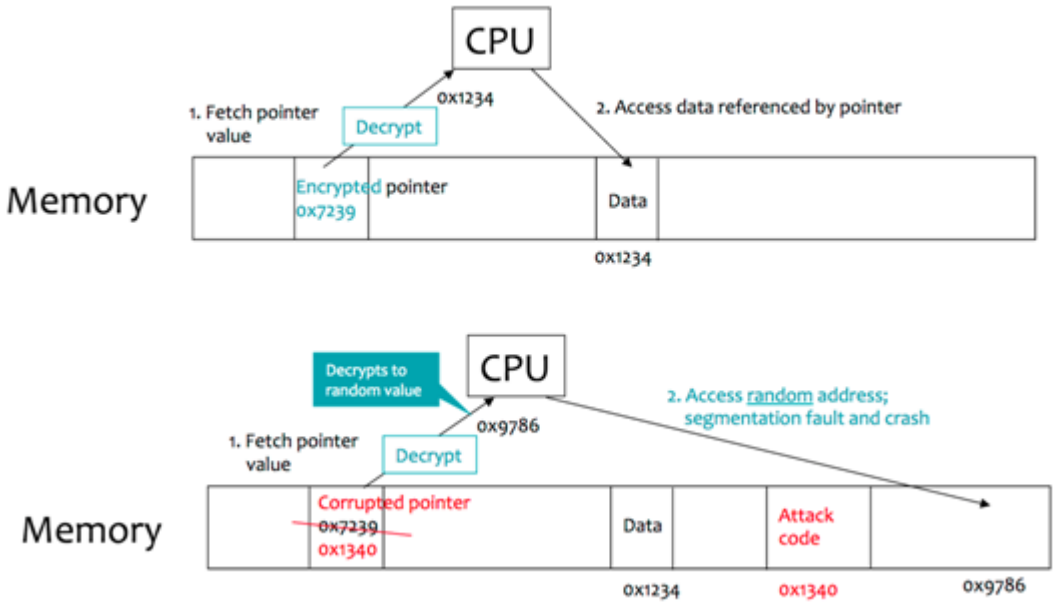

Figure 2.5: PointGuard dereference [13]

2.2.4 $W \oplus X$ & NX-bit

An adversary can inject malicious code into data section of memory and thereafter change control-flow of the program. Certain parts of memory can be made non-executable to avoid malicious code execution from data section. $W \oplus X$ is security feature used to protect memory which can be either writable or executable. NX-bit is used to divide memory into data section and code section.

Drawback: NX bit feature used to make some part of memory non-executable. NX bit feature can be bypassed by return-to-libc attack as it uses only executable code to trigger an attack.

Chapter 3

Related Work

There are many mitigation techniques that can be used against attacks such as buffer overflow, return-to-libc, ROP, etc. However, some of these techniques are impractical, and they are based on hardware. They do not provide complete protection from attacks. Stack canary technique is used to protect return address stored on the stack. However, an adversary can change the return address without changing canary. Over the years powerful attacks have been invented that make an adversary more powerful. Some mitigation techniques have high performance overhead. With the increase in security performance, overhead is also increased. CFI is the most promising technique against attacks such as buffer overflow. It enforces the program to follow the path defined by CFG. CFG consists of nodes which represent functions in a program. CFG is determined before program execution and applied during program execution. Forward edges of CFG represent function calls. During static analysis phase, destination of forward edges are computed, and they are enforced during program execution. CFI technique monitors program flow and ensures that control-flow of the program follows CFG. In recent years, Intel's architects have turned to control-flow enforcement technology (CET) [14].

3.1 CFI with Label Checking

Whenever there is a function call, the correct destination must be called. This technique uses labels which are randomly generated unique numbers. This technique considers three new instructions for CFI instrumentation. `Label ID` is used to add the label at the start of destination. `call ID DST` calls destination only if the destination has `Label ID` at the start of it's code. `ret ID` is used to return to caller [1].

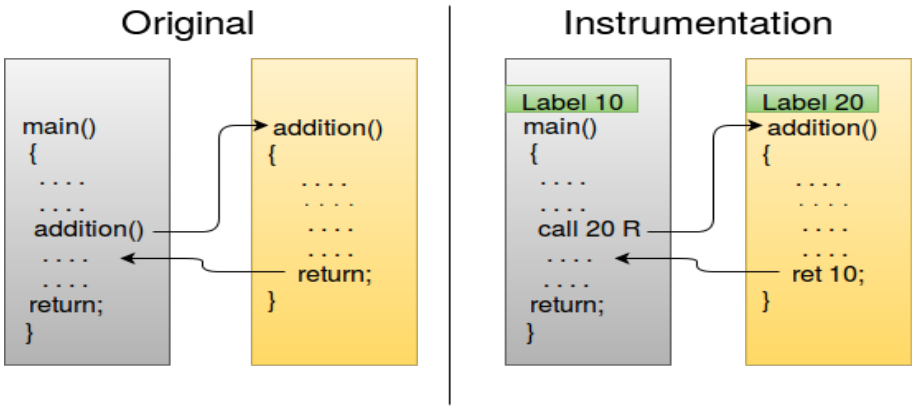

Figure 3.1: Label checking [1]

In Figure 3.1, `Label 10` and `Label 20` are assigned to function `main()` and `addition()` respectively. In original section, there is call from `main` to `addition`. After execution of `addition`, control returns to `main`. In instrumentation section, `call 20 R` is executed, label at destination, i.e., `20` and label present in call instruction are compared. When function returns, label of caller and label present in return instruction are compared. If comparison fails, it indicates CFI-failure.

Assumptions:

- Unique IDs: ID conflict may create problem while label checking. Therefore IDs used in CFI must be unique.
- Non writable code: If code memory is writable, CFI implementation code can be corrupted. Therefore code section memory must not be writable.
- Non executable DATA: If data section is executable, an adversary can add executable malicious code to data section.

Code Instrumentation [1]

	Assembly code	Comment
1	`jump ecx`	`; a jump instruction`

can be instrumented as:

```
1 cmp [ecx], 12345678h          ; compare data at destination
2 jne error_label               ; if not ID value, then fail
3 lae ecx, [ecx+4]              ; skip ID data at destination
4 jmp ecx                       ; jump to destination
```

In above code instrumentation, `jump ecx` is instrumented. `ecx` has destination address. `12345678h` is label added at the start of destination. If labels are matched control jumps to destination otherwise, it fails.

Drawback: Code instrumentation is not possible all the time. An adversary can change the return address stored on the stack and hijack control-flow of the program. Therefore this technique does not ensure the integrity of control-flow of the program. This technique does not have protection against buffer overflow attack.

3.2 CCFIR (Compact Control Flow Integrity and Randomization)

This mitigation technique is used to secure function pointer and return address. CCFIR [8] performs enforcement by redirecting control-flow transfer through new code section called "Springboard" which works as mediator or controller. Whenever there is control-flow jump, it forces control-flow to pass through Springboard section. An adversary can not insert jump in the middle of instructions.

In Figure 3.2, in the original section, function `foo` is called, and control returns to next line of `call foo`. In the Hardened section, redirection is implemented through Springboard. Control-flow is directed to Springboard and returned through this section only.

Drawback: It has a compatibility issue. Figure 3.3 shows call from a protected module to an unprotected module. Once control reaches to an unprotected module, the application can no longer remain secure through Springboard. Rewriting every module is not possible all the time.

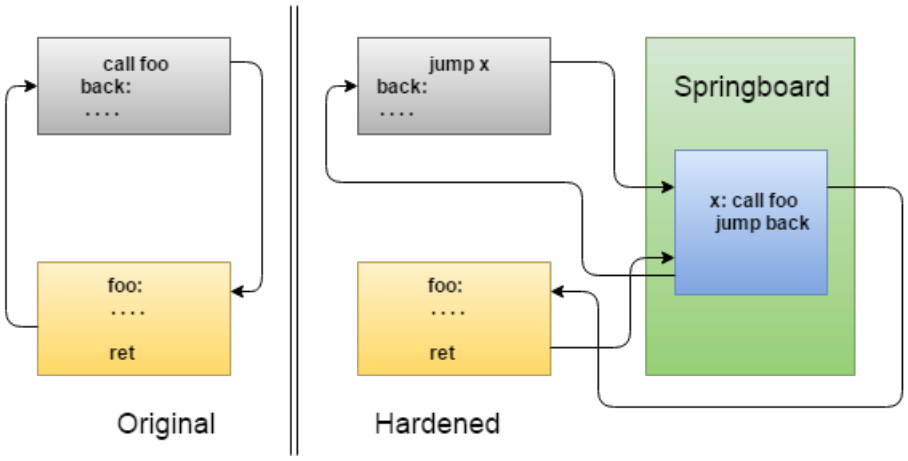

Figure 3.2: CCFIR technique [8]

3.3 CCFI (Cryptographically-enforced Control Flow Integrity)

CCFI [7] protects function pointer and return address with the help of Message Authentication Code (MAC). Following function is used to compute MAC.

```
MAC(K, pointer, class)
```

MAC function consists of three parameters: `K` is a secret key, `pointer` is function pointer or return address, and `class` is a flag to identify between a function pointer or return address. MAC is used to protect the return address saved on the stack. MAC of the return address is computed when the return address is stored on the stack. When a function returns, MAC is re-calculated and compared with stored MAC. Both MACs must match. If an adversary changes the return address stored on the stack, re-calculated MAC and stored MAC does not match, and it indicates an error.

Drawback: As shown in Figure 3.4, If an adversary knows old function pointer and MAC, the return address and MAC stored on the stack can be overwritten with the old value of the return address and MAC.

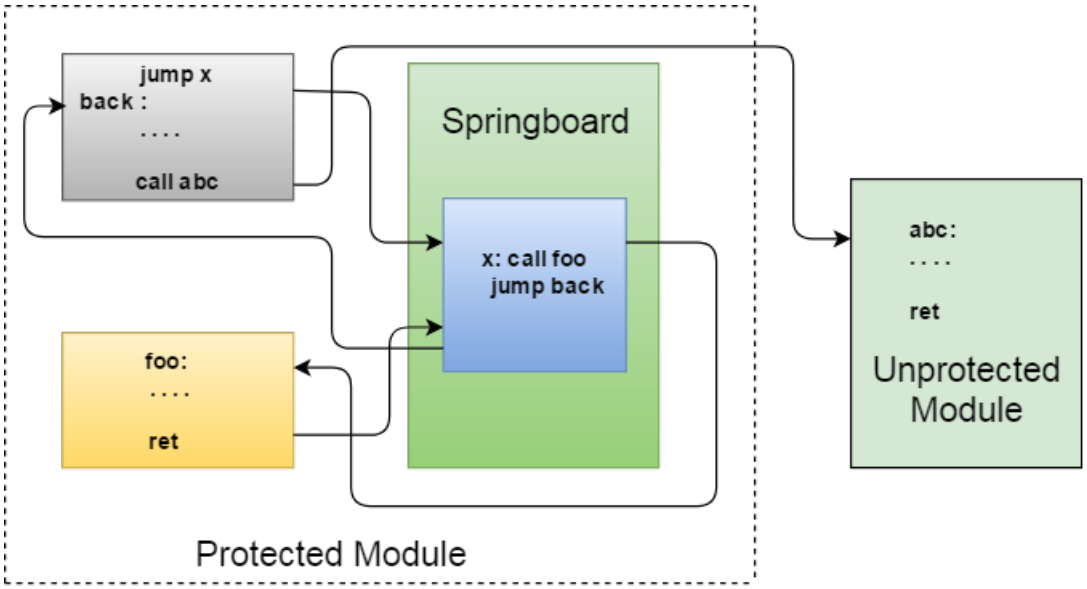

Figure 3.3: CCFIR drawback [8]

3.4 CFCSS (Control Flow Checking by Software Signature)

CFCSS [9] one of the mitigation techniques which uses software signature. The program is divided into blocks called nodes. Each node is assigned a unique number called signature (s). The signature difference (d) is XOR of signature of a source (caller) and destination (callee) node. During compile-time, the signature difference is calculated and assigned to each node. Runtime signature (G) is XOR of signature difference (d) and signature of a source node. During runtime, G is calculated and compared with the signature of destination node. G and signature of destination node must match for correct flow between a source node and destination node.

Figure 3.6 shows, V1 to V2 is correct branching as G2 and s2 are matched. V1 to V4 is illegal branch as G4 is not matched with s4.

Drawback: A unique signature is assigned to each node of the program. During compile-time, the signature difference (d) is computed and assigned to each node. Figure 3.6 shows V1 and V3 have a different signature. Therefore there are two values of d assigned to V5. To assign a single value of d to V5, the same signature must be assigned to V1 and V3. This violates the constraint of software signature.

Figure 3.6 shows, V5 has two values of signature difference (d). To assign single signature difference (d) to V5, signature of V1 and V3 must be same.

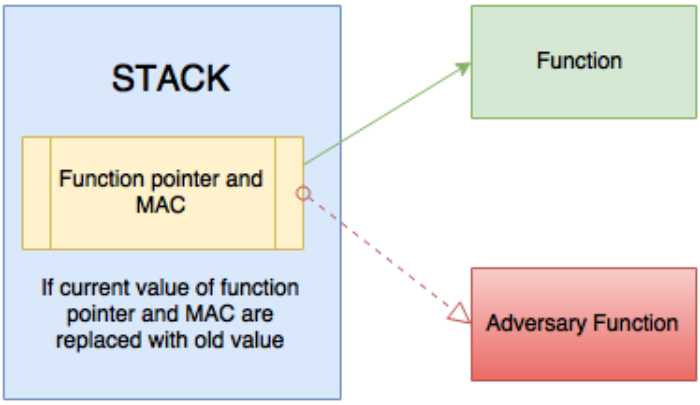

Figure 3.4: CCFI MAC overwritten [7]

3.5 IFCC (Indirect Function Call Check)

IFCC [6] protects indirect calls by generating jump table. Jump table consists of jump instructions to function. Instrumentation replaces each function with the address of the corresponding entry in the jump table.

Drawback: The address of jump table is stored on the stack. The address of jump table can be overwritten by bogus jump table and adversary can execute the malicious function.

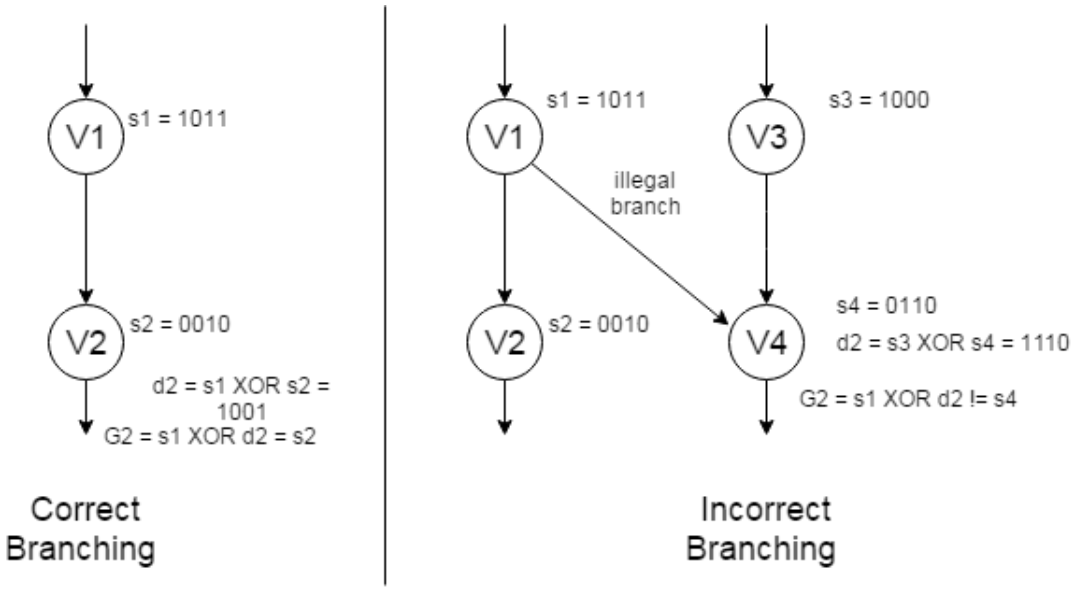

Figure 3.5: CFCSS technique[9]

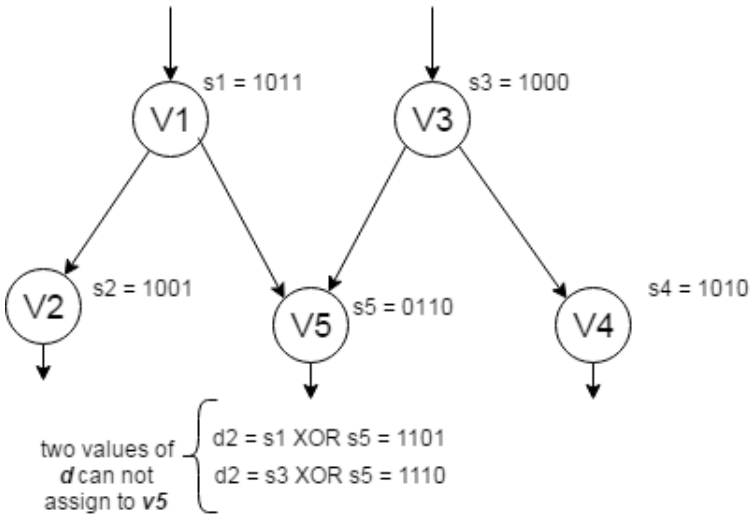

Figure 3.6: CFCSS branch-in-fan [9]

Chapter 4

Our CFI Implementation

There are many CFI techniques that include Control Flow Checking with Label Checking, Control Flow Checking by Software Signature (CFCSS) [9], Compact Control Flow Integrity and Randomization (CCFIR) [8], Cryptographically-enforced Control Flow Integrity (CCFI) [7], Indirect Function Call Check (IFCC) [6], etc. However, these CFI techniques have practical limitations and performance overhead. Many of these CFI techniques do not have protection against buffer overflow attack. Some of these do not work in case of recursive calls (a function calling itself).

Our approach of CFI implementation mainly includes generating regular expression of a program which represents the CFG. The first step of implementation is to derive CFG of a program in the form of DOT (Graph Descriptive Language) file. We have used egypt-tool to generate CFG of a program. Regular expression of the program is generated from CFG. Regular expression of the graph is derived using Brzozowski's algorithm [11]. We maintain execution path of a program. Execution paths that satisfy the regular expression are considered the valid paths of execution. Execution paths that do not satisfy the regular expression are considered as the invalid paths of execution. All the invalid paths generate an error as they do not satisfy regular expression. We also check integrity of the return address stored on the stack. We maintain a replica of the stack consists of the return address. An adversary can hijack control-flow of the program by changing return address stored on the stack. Checking integrity of the return address stored on the stack helps to protect control-flow integrity of a program.

We consider complete execution path of the program for CFI implementation. Undefined change in the path of execution by an adversary can be detected with the help of regular expression. Our CFI implementation mainly includes: deriving

CFG of a program, deriving a regular expression of a program, and checking for CFI violation.

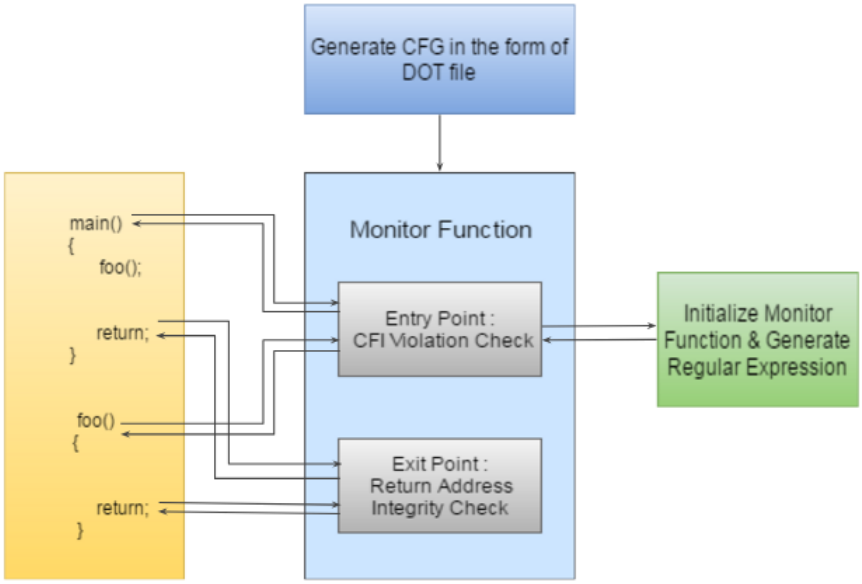

Figure 4.1: CFI implementation view

4.1 Derive CFG of a Program

We have used `egypt-tool (version 1.1.0)` [16] for deriving CFG of a program. Below mentioned command is used to generate the graph of `main.c` program.

```
gengraph -o graph -t png main.c
```

Program: `main.c`

```
1  #include<stdio.h>
2  int Sub(int x, int y){
3      int z = x - y;
4      Display(z);
5      return 0;
6  }
7  int Power(int x, int y){
8      Addition(x, y);
9      Sub(x, y);
10     return 0;
11 }
12 int Addition(int x, int y){
13     return x + y;
14 }
15 int Display(int x){
16     printf("%d", x);
17     return 0;
18 }
19 int Division(int x, int y){
20     int z = x / y;
21     Display(z);
22     return z;
23 }
24
25 int main(void){
26     Division(40, 5);
27     Addition(2, 4);
28     Power(2, 5);
29     return 0;
30 }
```

Output: dot (Graph Descriptive Language) file

```
1 digraph callgraph {
2     "Power" -> "Addition" [style=solid];
3     "Power" -> "Sub" [style=solid];
4     "Division" -> "Display" [style=solid];
5     "main" -> "Power" [style=solid];
6     "main" -> "Addition" [style=solid];
7     "main" -> "Division" [style=solid];
8     "Sub" -> "Display" [style=solid];
9 }
```

Command to convert dot file to pn file is shown follow below.

```
dot -Tps *.dot -o *.pn
```

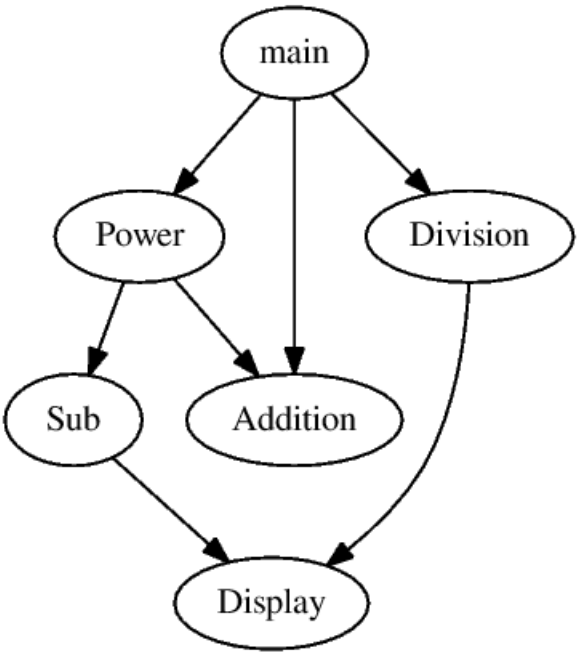

Figure 4.2: CFG of `main.c`

Figure 4.3 shows, matrix list form of CFG shown in 4.2. In each chain, the first node is a caller, and subsequent nodes are callee.

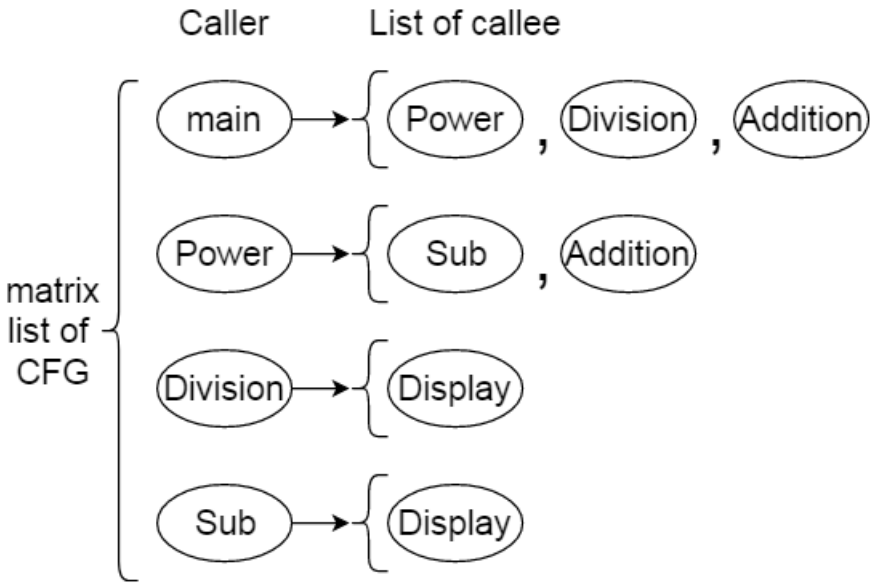

Figure 4.3: Matrix list of CFG

Following commands are used to run `main.c` program with CFI check.

```
1.  gcc -finstrument-functions -std=c++0x -c -o main.o main.c
2.  g++ -g -rdynamic -std=c++0x -c -o trace.o func.cpp
3.  g++ -g -rdynamic -std=c++0x trace.o main.o -o cfimonitor
4.  ./cfimonitor
```

4.2 Derive Regular Expression of a Program

We have used Brzozowski's Algorithm [11] to generate a regular expression. This algorithm is an algebraic method. It is divided into two parts: the first part is to convert CFG to Deterministic Finite Automata (DFA) and the second part is to convert DFA to a regular expression. It creates a system of linear equations. A regular expression can be obtained by solving the system of linear equations. Following Brzozowski's Algorithm shows conversion from DFA to a regular expression.

Program: CFG to DFA

```
for i in range (0, n)
        for j in range (0, n)
                A[i, j] <- 0
        end for
        B[i] <- 0
end for
for i in range (0, n-1)
        if node i represents an entry block then
                A[0, i+1] <- node i
        end if
        if node i represents an exit block then
                B[i+1] <- e
        end if
end for
for i in range (0, n-1)
        for j in range (0, n-1)
                if there is an edge from node i to node j
                        A[i+1, j+1] <- node j
                end if
        end for
end for
```

Program: DFA to Regular Expression

```
for k in range(0, n)
        B[k] <- (star(A[k, k])) B[k]
                for j in range(0, k-1)
                        A[k, j] <- (star(A[k, k])) * A[k, j]
                end for
        for i in range(0, k-1)
                B[i] <- B[i] + A[i, k] * B[k]
                for j in range(0, k-1)
                        A[i, j] <- A[i, j] + A[i, k] * A[k, j]
                end for
        end for
        for i in range(0, k-1)
                A[i, k] <- 0
        end for
end for
```

Following regular expression will be generated for CFG shown in Figure 4.4.

```
Regular Expression:  main(Addition | Division.Display)
```

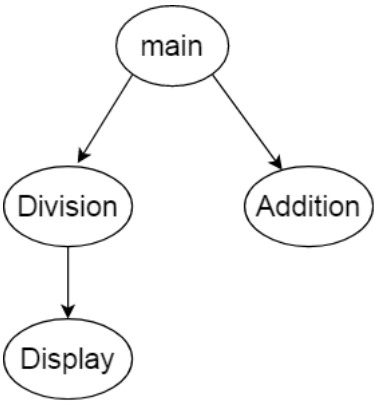

Figure 4.4: CFG of regular expression `main(Addition | Division.Display)`

All the strings, which satisfy regular expression, are considered the valid paths of execution. The valid paths of execution for `main(Addition | Division.Display)` regular expression includes `main->Addition`, `main->Division->Display`, etc.

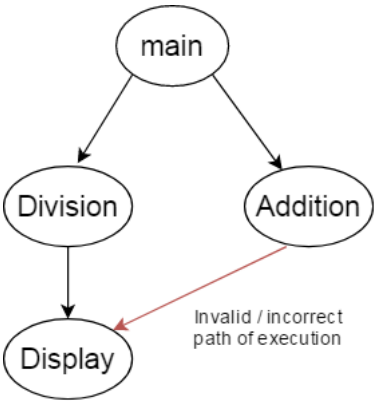

Figure 4.5: Invalid path of execution

As shown in Figure 4.5, `main->Addition->Display` is considered as invalid path of execution since it does not satisfy regular expression `main(Addition | Division.Display)`. Figure 4.5 shows red arrow which represents undefined jump from `Addition()` to `Display()` within CFG.

4.3 Defining CFI Violation

CFI violation is defined as an undefined change in the control-flow of execution or calling a function that is not a part of the program. Execution path which does not satisfy regular expression is invalid path of control-flow.

CFI enforces control-flow of the program to follow defined path. These paths of execution can be defined with CFG. We first derive CFG of a program which is in the form of DOT (Graph Descriptive Language) file.

An undefined control-flow jump, which is not a part of CFG, is detected with the help of regular expression. This strengthens the security of control-flow of a program.

While returning from called function, we check the integrity of return address stored on the stack. An adversary can overwrite the return address of a function. Change in return address may lead to the hijacking of a control-flow of the program. Checking return address on the stack ensures the correct flow of the program.

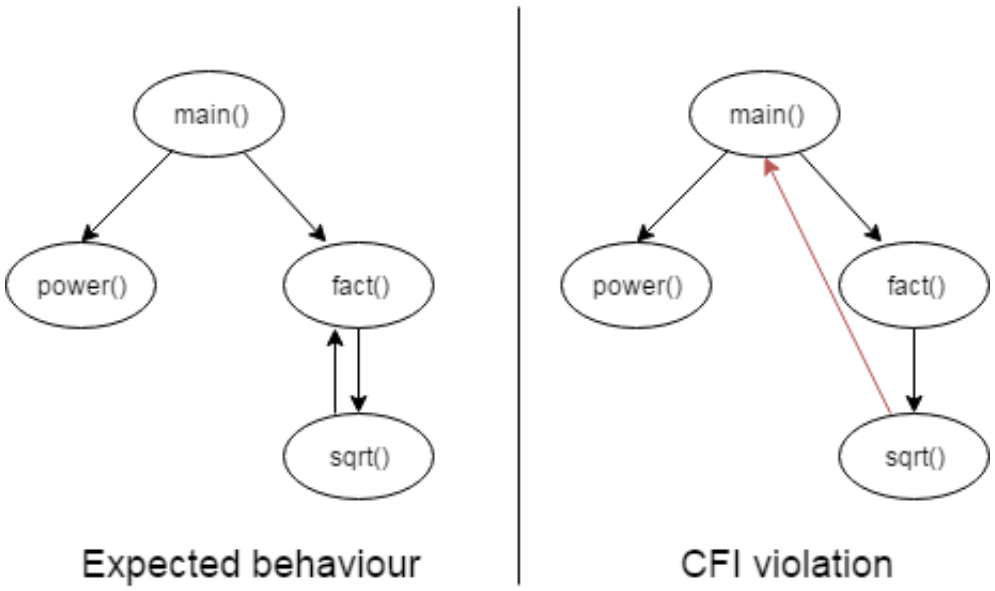

Figure 4.6: CFI violation in return call

Figure 4.6 shows CFI violation while returning from function `sqrt()` which is called from `fact()`. Program control is supposed to return to correct caller that is function `fact()`. However, an adversary can change the return address and direct the program flow to continue from `main()` function.

This CFI violation can be detected by checking return address stored on the stack. Before `sqrt()` is called, we store the return address. We compare return

address with the stored return address. This can detect a change in the return address on the stack. We maintain a replica of the stack which consists of only return address to detect a change in the return address. This helps to check the integrity of return address stored on the stack.

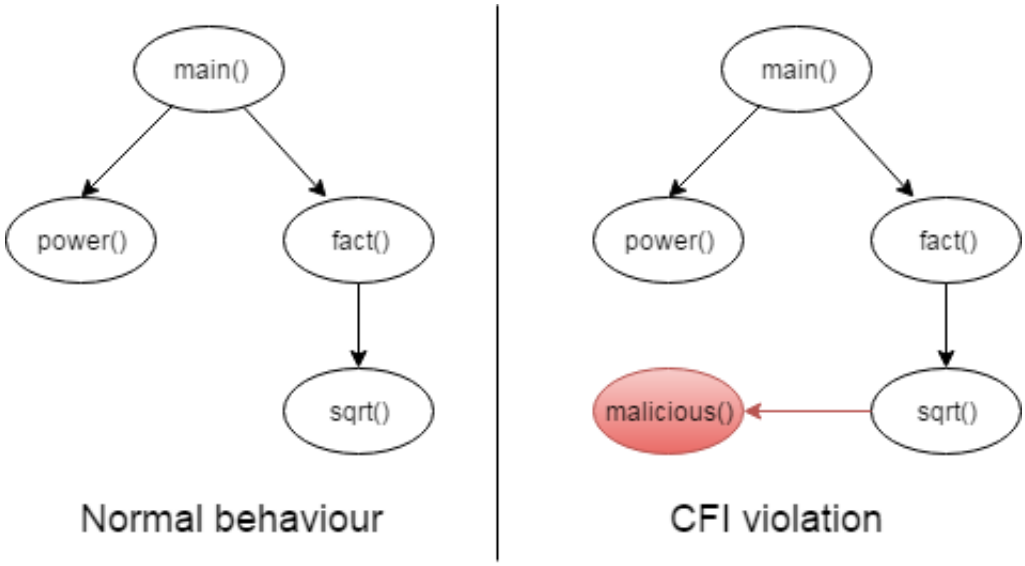

Figure 4.7: CFI violation in function call

Figure 4.7 shows undefined jump from `sqrt()` function to adversary function `malicious()` which is not a part of program. `sqrt()` is supposed to return to `fact()`. However an adversary changes return address to address of `malicious()` function. When control reaches to function `malicious()`, before it's execution starts, regular expression detects undefined control-flow of the program. Regular expression takes care of all the undefined control-flow paths.

4.4 Monitor Function

We have implemented a monitoring functionality that observes all changes in the control-flow of the program. Whenever a function call or a return statement is executed, this functionality kicks into action.

- Before program execution starts, monitor function is initialized. It converts DOT file data into matrix list. Following map is used to store mapping between caller and callee.

  ```
  map <string, list<string> > callee_graph;
  ```

  `callee_graph` is used to store CFG in matrix list form. First parameter of `callee_graph` is caller function and second parameter of `callee_graph` is list of callee.

- For complex CFG, regular expression generated will be long. We have assigned a unique number to each function node of CFG. A regular expression consists of numbers assigned to functions instead of the function names. This reduces a length of regular expression. Following map is used for mapping between the function name and it's unique number.

  ```
  map <string, int> funcname_to_funcnum;
  ```

- The last step of monitor function initialization is to generate a regular expression. Brzozowski's algorithm [11] is used to generate a regular expression.

4.5 Function Instrumentation

This is a part of monitor function. When function call statement is executed, before callee is executed, following function is called.

`void __cyg_profile_func_enter(void*, void*);` [15]

In `__cyg_profile_func_enter()` [15] function, callee function name is fetched using `backtrace_symbols()` [17] function. The function name is converted into it's unique number using `funcname_to_funcnum` map. We maintain `function_call_chain` using these unique numbers of callee functions. This `function_call_chain` must satisfy regular expression. This indicates control-flow of a program is within CFG. `regex_match(func_call_chain, txt_regex)` function is used to check whether `function_call_chain` satisfies regular expression.

`void __cyg_profile_func_exit(void*, void*);` [15]

Before callee function returns to caller, `__cyg_profile_func_exit()` [15] function is called. In `__cyg_profile_func_exit()` [15] function, `__builtin_return_address(1)` function is used to get the return address stored on the stack. We maintain replica of the stack which consists of return address. whenever a function is called, we maintain return address in replica of the stack. when function returns, we compare return address stored on replica of the stack with return address stored on the actual stack to check CFI violation.

Chapter 5

CFI Violation and Detection

CFI violation is defined as an undefined change in the control-flow of execution or calling a function that is not a part of the program. Here we discuss four scenarios of CFI violation and detection.

5.1 Scenario 1: Buffer Overflow

In Buffer_overflow.c program, an adversary is trying to corrupt return address stored on the stack for function `foo()`. `foo()` function has buffer in its activation record. An adversary is using buffer to corrupt return address of `foo()` stored on stack.

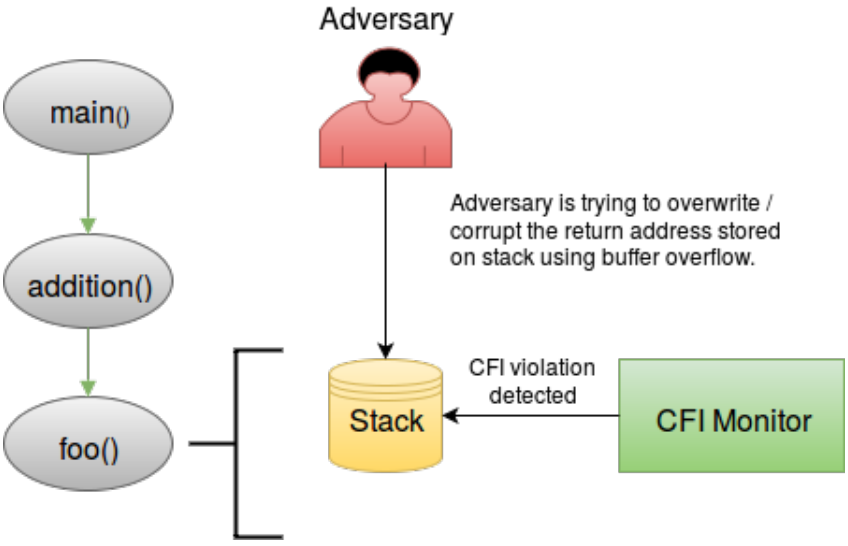

Figure 5.1: Buffer overflow.

Example 1: Buffer_overflow.c

```
1  #include<stdio.h>
2  void abc(void){}
3
4  void foo(void){
5       void (*buffer[12])(void);
6       int k;
7       for(k = 0 ; k < 128 ; k++)
8            buffer[k] = abc;
9  }
10 int addition(void){
11      foo();
12      return 0;
13 }
14 int main(void){
15      addition();
16      return 0;
17 }
```

```
(gdb)
Dump of assembler code for function addition:
   0x00000000004004f9 <+0>:     push   %rbp
   0x00000000004004fa <+1>:     mov    %rsp,%rbp
   0x00000000004004fd <+4>:     callq  0x4004bc <foo>
   0x0000000000400502 <+9>:     mov    $0x0,%eax
   0x0000000000400507 <+14>:    pop    %rbp
   0x0000000000400508 <+15>:    retq
End of assembler dump.
```

Figure 5.2: Buffer_overflow.c: assembly code of `addition()` function

```
(gdb) x/20wx $sp
0x7fffffffe4e0: 0xffffe4f0      0x00007fff      0x00400502      0x00000000
0x7fffffffe4f0: 0xffffe500      0x00007fff      0x00400512      0x00000000
0x7fffffffe500: 0x00000000      0x00000000      0xf7a52b45      0x00007fff
0x7fffffffe510: 0x00000000      0x00000000      0xffffe5e8      0x00007fff
0x7fffffffe520: 0x00000000      0x00000001      0x00400509      0x00000000
```

Figure 5.3: Buffer_overflow.c: stack content

Figure 5.2 shows assembly code for `addition()` function. `400502` is return address of function `foo()`. Figure 5.3 shows return address of function `foo()` stored on the stack.

```
**************Enter*****************
addition->foo
foo->abc
main->addition
.....................................
abc->3
addition->1
foo->2
main->0
.....................................

Regular Expresion of Graph:0(|1(|2(|3)))
.....................................

Monitor function initialization completed
.....................................
func_call_chain: 0
func_call_chain is matched with Regular Expression

**************Enter*****************
func_call_chain: 01
func_call_chain is matched with Regular Expression

**************Enter*****************
func_call_chain: 012
func_call_chain is matched with Regular Expression

++++++++++++++++Exit++++++++++++++++++
Error: Return address is changed
```

Figure 5.4: Buffer_overflow.c: CFI violation detection

Figure 5.4 shows execution of Buffer_overflow.c with our CFI check. The program execution starts with `main()`. Before `main()` executes, monitor function is initialized. Each function name is assigned a unique number. Following regular expression is generated.

`Regular Expression for Buffer_overflow.c: 0(|1(|2(|3)))`

When `main()` function starts it's execution `function_call_chain` becomes `0`. Function `main()` calls `addition()`. Therefore `function_call_chain` becomes `01`. `addition()` calls `foo()`. `function_call_chain` becomes `012`. Return address of `foo()` is corrupted using buffer overflow. This causes error message (`Error: Return address is changed`) and program termination.

5.2 Scenario 2: Buffer Overflow with Recursion

Figure 5.6 shows regular expression for Buffer_overflow_with_recursion.c program. Execution starts with `main()`. `foo()` function is recursively called. The return address is changed when value of x reaches to 0 using buffer overflow attack.

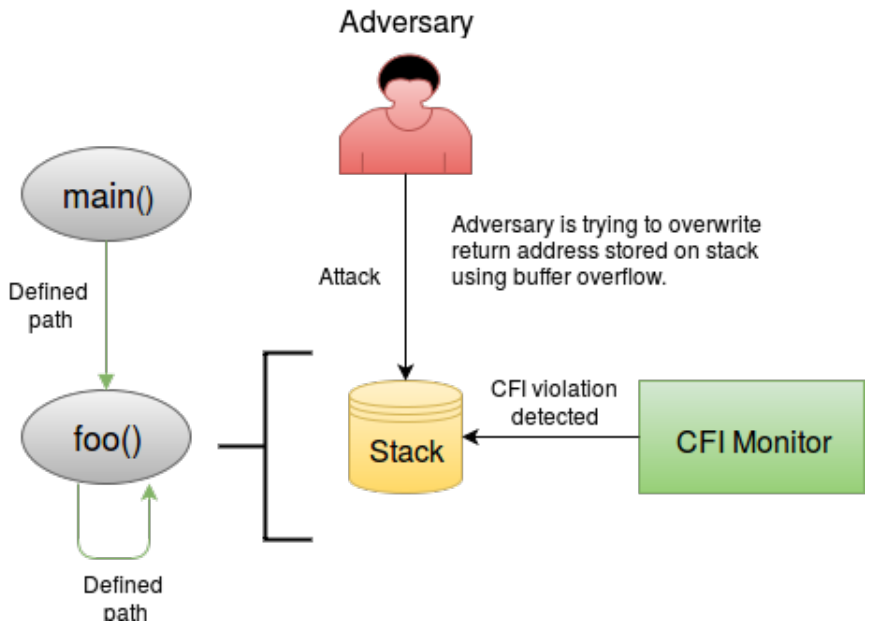

Figure 5.5: Buffer overflow with recursion

Example 2: Buffer_overflow_with_recursion.c

```
1  #include<stdio.h>
2  void foo(int x){
3      if(x == 0)
4      {
5          int k;
6          char buffer[12];
7          for(k = 0 ; k < 128 ; k++)
8              buffer[k] = '0';
9      }
10     if(x>0)
11         foo(x-1);
12 }
13 int main(void){
14     foo(2);
15     return 0;
16 }
```

```
***************Enter*****************
foo->foo
main->foo
.....................................
foo->1
main->0
.....................................

Regular Expresion of Graph:0(|11*)
.....................................

Monitor function initialization completed
.....................................
func_call_chain: 0
func_call_chain is matched with Regular Expression

***************Enter*****************
func_call_chain: 01
func_call_chain is matched with Regular Expression

***************Enter*****************
func_call_chain: 011
func_call_chain is matched with Regular Expression

***************Enter*****************
func_call_chain: 0111
func_call_chain is matched with Regular Expression

+++++++++++++++Exit++++++++++++++++++
Error: Return address is changed
```

Figure 5.6: Buffer_overflow_with_recursion.c: output

5.3 Scenario 3: Jump to Adversary Function

In Adversary_function_without_cfi.c program, activation record of `foo()` has `pointer` (int type), `buffer`, and `buffer1`. Difference between return address of `foo()` stored on the stack and address of `buffer1` is `56`. Therefore we have incremented `pointer` by `56` to excess return address of `foo()`. Once `pointer` is pointing to the return address stored on the stack. we have changed the return address of `foo()` to address of `attack()` function.

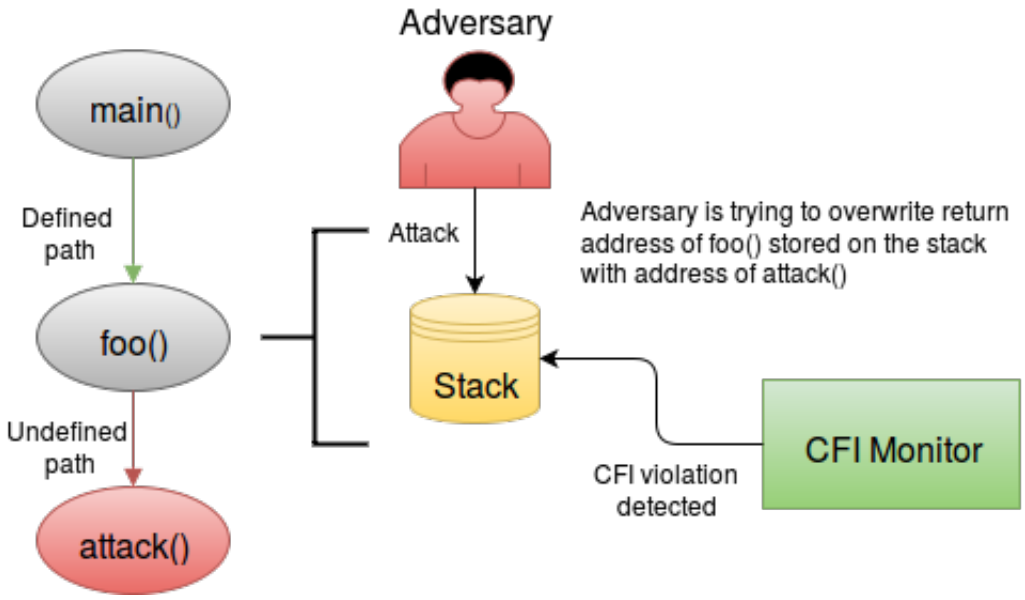

Figure 5.7: Jump to adversary function

Example 3.1: Adversary_function_without_cfi.c

```
1  void attack(void){
2      printf("Attacked");
3  }
4  void foo(int a, int b){
5      int *pointer;
6      char buffer[5];
7      int c = b + a;
8      char buffer1[10];
9      pointer = buffer1 + 56;
10     printf("Return address stored on stack: %p", *pointer);
11     *pointer = 4195655;
12 }
13 int main(void){
14     printf("In main");
15     foo(4, 5);
16     printf("end");
17     return 0;
18 }
```

```
In main
Return address:0x4005bb
Attacked:
```

Figure 5.8: Adversary_function_without_cfi.c: output

```
(gdb)
Dump of assembler code for function foo:
   0x0000000000400556 <+0>:     push   %rbp
   0x0000000000400557 <+1>:     mov    %rsp,%rbp
   0x000000000040055a <+4>:     sub    $0x40,%rsp
   0x000000000040055e <+8>:     mov    %edi,-0x34(%rbp)
   0x0000000000400561 <+11>:    mov    %esi,-0x38(%rbp)
   0x0000000000400564 <+14>:    mov    -0x38(%rbp),%edx
   0x0000000000400567 <+17>:    mov    -0x34(%rbp),%eax
   0x000000000040056a <+20>:    add    %edx,%eax
   0x000000000040056c <+22>:    mov    %eax,-0x4(%rbp)
   0x000000000040056f <+25>:    lea    -0x30(%rbp),%rax
   0x0000000000400573 <+29>:    add    $0x38,%rax
   0x0000000000400577 <+33>:    mov    %rax,-0x10(%rbp)
   0x000000000040057b <+37>:    mov    -0x10(%rbp),%rax
   0x000000000040057f <+41>:    mov    (%rax),%eax
   0x0000000000400581 <+43>:    mov    %eax,%esi
   0x0000000000400583 <+45>:    mov    $0x40065e,%edi
   0x0000000000400588 <+50>:    mov    $0x0,%eax
   0x000000000040058d <+55>:    callq  0x400420 <printf@plt>
   0x0000000000400592 <+60>:    mov    -0x10(%rbp),%rax
   0x0000000000400596 <+64>:    movl   $0x400547,(%rax)
   0x000000000040059c <+70>:    leaveq
   0x000000000040059d <+71>:    retq
End of_assembler dump.
```

Figure 5.9: Adversary_function_without_cfi.c: buffer1 address

Figure 5.9 shows buffer1 is stored at 30 bytes below stack pointer. Frame pointer is 8 bytes. Therefore `buffer1` and return address has difference of 38 bytes. 38 is hexadecimal value. If we convert it into decimal, it is 56. Therefore 56 is added to buffer1 to point to return address.

```
(gdb)
Dump of assembler code for function main:
   0x000000000040059e <+0>:     push   %rbp
   0x000000000040059f <+1>:     mov    %rsp,%rbp
   0x00000000004005a2 <+4>:     mov    $0x400670,%edi
   0x00000000004005a7 <+9>:     callq  0x400410 <puts@plt>
   0x00000000004005ac <+14>:    mov    $0x5,%esi
   0x00000000004005b1 <+19>:    mov    $0x4,%edi
   0x00000000004005b6 <+24>:    callq  0x400556 <foo>
   0x00000000004005bb <+29>:    mov    $0x400678,%edi
   0x00000000004005c0 <+34>:    callq  0x400410 <puts@plt>
   0x00000000004005c5 <+39>:    mov    $0x0,%eax
   0x00000000004005ca <+44>:    pop    %rbp
   0x00000000004005cb <+45>:    retq
End of assembler dump.
```

Figure 5.10: Adversary_function_without_cfi.c: assembly code of function `main()`

Figure 5.10 shows return address of `foo()` which is next instruction to be executed after `foo()` returns. Figure 5.7 shows program execution starts with `main()`. There is call to function `foo()` from `main()`. The return address of `foo()` is overwritten with address of `attack()` function. Therefore program control does not return to `main()`. `attack()` function is executed instead of returning to `main()`. `attack()` is not a part of CFG. This is undefined control-flow of the program.

```
(gdb) x/20wx $sp
0x7fffffffe510: 0xffffe520      0x00007fff      0x004005bb      0x00000000
0x7fffffffe520: 0x00000000      0x00000000      0xf7a52b45      0x00007fff
0x7fffffffe530: 0x00000000      0x00000000      0xffffe608      0x00007fff
0x7fffffffe540: 0x00000000      0x00000001      0x0040059e      0x00000000
0x7fffffffe550: 0x00000000      0x00000000      0x9354dbf1      0xfee2fa88
(gdb)
```

Figure 5.11: Adversary_function_without_cfi.c: stack content

Figure 5.11 shows the return address stored on the stack.

Example 3.2: Adversary_function_with_cfi.c

```
1  void attack(void){
2      printf("Attacked");
3  }
4  void foo(int a, int b){
5      int *pointer;
6      char buffer[5];
7      int c = b + a;
8      char buffer1[10];
9      pointer = buffer1 + 56;
10     printf("Return address stored on stack: %p", *pointer);
11     *pointer = 4714583 ;
12 }
13 int main(void){
14     printf("In main");
15     foo(4,5);
16     printf("end");
17     return 0;
18 }
```

In Adversary_function_with_cfi.c program, value assigned to pointer is different as Adversary_function_with_cfi.c is compiled with monitor function to check cfi violation.

```
***************Enter*****************
main->foo
.....................................
foo->1
main->0
.....................................

Regular Expresion of Graph:0(|1)
.....................................

Monitor function initialization completed
.....................................
func_call_chain: 0
func_call_chain is matched with Regular Expression

***************Enter*****************
func_call_chain: 01
func_call_chain is matched with Regular Expression

+++++++++++++++Exit++++++++++++++++++
Error: Return address is changed
```

Figure 5.12: Adversary_function_with_cfi.c: output

5.4 Scenario 4: Skipping Function Call

In Skip_function_call.c program, return address is increased to skip function call statement `sub(3, 4)`. Control-flow of the program must follow defined path of execution. Skipping function call is one the CFI violations.

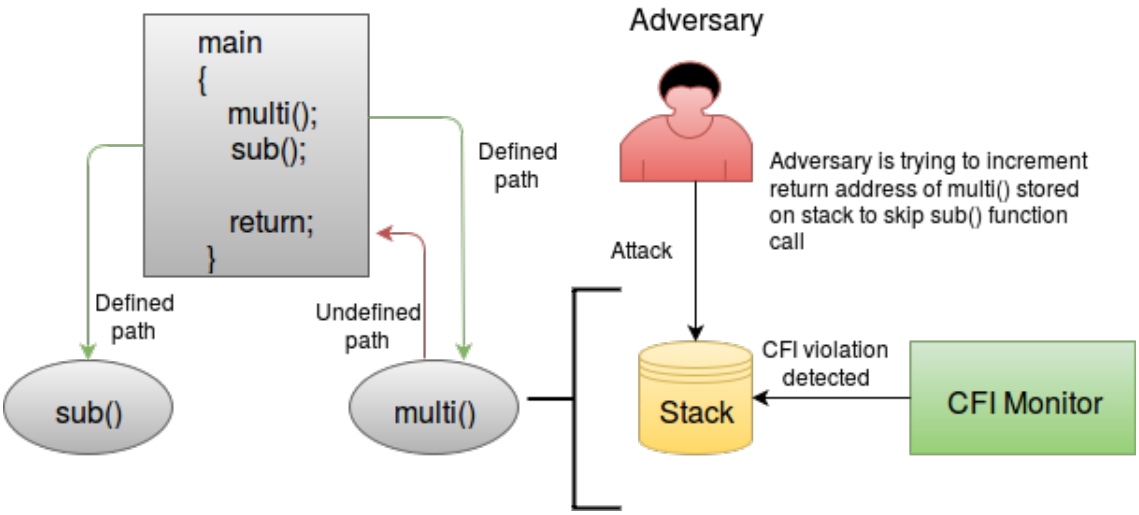

Figure 5.13: Skipping function call

Example 4: Skip_function_call.c

```
1  void sub(int x, int y){
2      printf("In sub");
3      int b=0;
4  }
5  void multi(int a, int b){
6      int *pointer;
7      char buffer1[5];
8      int c = a + b;
9      char buffer2[10];
10     pointer = buffer2 + 56;
11     printf("Return Address: %p",*pointer);
12     printf("In multi");
13     *pointer = *pointer + 15;
14 }
15 int main(void){
16     printf("In main");
17     multi(2,3);
18     sub(3,4);
19     return 0;
20 }
```

.

```
In main
Return address: 0x4005db
In multi
```

Figure 5.14: Skip_function_call.c: output

In Skip_function_call.c program, return address on stack is increased by `15` to skip function call `sub(3, 4)`. After `multi()` function execution, control should reach to `sub()` function for execution. `sub()` function execution is skipped by changing return address.

```
***************Enter*****************
main->sub multi
.....................................
main->0
multi->2
sub->1
.....................................

Regular Expresion of Graph:0((|2)|1)
.....................................

Monitor function initialization completed
.....................................
func_call_chain: 0
func_call_chain is matched with Regular Expression

In main
***************Enter*****************
func_call_chain: 02
func_call_chain is matched with Regular Expression

Return address: 0x47f0f3
In multi
+++++++++++++++Exit++++++++++++++++++
Error: Return address is changed
```

Figure 5.15: Skip_function_call.c: output with CFI check

Chapter 6

Conclusion & Future Work

CFI is a simple and trustworthy mitigation technique against attempts to hijack program's execution path. We have implemented a CFI based technique that ensures integrity of control-flow of a program using a regular expression and by checking the integrity of return address stored on the stack. Overhead is always important parameter for any technique however our main objective had been to guarantee CFI. Our work is useful for programs that are written in C & C++. Similar approach can be easily extended to other programming environments.

References

[1] Martin Abadi, Mihai Budiu, Ulfar Erlingsson, Jay Ligatti. Control-flow Integrity. In *Proceedings of the 12th ACM conference on Computer and communications security*, CCS, 2005.

[2] Mauro Conti, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, Ahmad-Reza Sadeghi. Losing Control: On the Effectiveness of Control-Flow Integrity under Stack Attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015.

[3] Tyler Bletsch, Xuxian Jiang, Vince Freeh. Mitigating Code-Reuse Attacks with Control-Flow Locking. *In Proceedings of the 27th Annual Computer Security Applications Conference*, 2011.

[4] Mark M. Seege. Using Control-Flow Techniques in a Security Context: A Survey on Common Prototypes and Their Common Weakness. In *Proceedings of the 2011 International Conference on Network Computing and Information Security*, 2011.

[5] Thurston H. Y. Dang, Petros Maniatis, David Wagner. The Performance Cost of Shadow Stacks and Stack Canaries. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, CCS, 2015.

[6] Tice, Caroline, et al. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. *USENIX Security Symposium*, 2014.

[7] Ali Jose Mashtizadeh, Andrea Bittau, David Mazieres, Dan Boneh. Cryptographically Enforced Control Flow Integrity. In *proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015.

[8] Chao Zhang, Tao Wei, Zhaofeng Chen, Lei Duan, Laszlo Szekeres, Stephen McCamant, Dawn Song, Wei Zou. Practical Control Flow Integrity and Randomization for Binary Executables. In *proceeding SP '13 Proceedings of the 2013 IEEE Symposium on Security and Privacy*, 2013.

[9] N. Oh, P. P. Shirvani and E. J. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 2002.

[10] Benso, Alfredo, et al. Control-flow checking via regular expressions. *Test Symposium, 2001. Proceedings. 10th Asian*, IEEE, 2001.

[11] Sibel Toprak. Intraprocedural Control Flow Visualization based on Regular Expressions. *Hamburg University of Technology (TUHH)*, 17 Jan 2014.

[12] Ross J. Anderson. 2008. *A Guide to Building Dependable Distributed Systems (2 ed.)*. Wiley Publishing.

[13] Cowan, Crispin, et al. PointGuard TM: protecting pointers from buffer overflow vulnerabilities. *Proceedings of the 12th conference on USENIX Security Symposium.* Vol. 12. 2003.

[14] Hardware is the new software
`https://www.microsoft.com/en-us/research/wp-content/uploads/2017/05/baumann-hotos17.pdf`

[15] Function Instrumentation
`https://gcc.gnu.org/onlinedocs/gcc/Instrumentation-Options.html`

[16] egypt tool
`https://www.gson.org/egypt/`

[17] Backtrace
`https://www.gnu.org/software/libc/manual/html_node/Backtraces.html`

[18] Return-to-libc
`https://en.wikipedia.org/wiki/Return-to-libc_attack`

[19] Return-oriented_programming
`https://en.wikipedia.org/wiki/Return-oriented_programming`

[20] Smashing the stack for fun and profit
http://phrack.org/issues/49/14.html

[21] Exploiting format string vulnerabilities
https://crypto.stanford.edu/cs155/papers/formatstring-1.2.pdf

[22] Once upon a free()
http://phrack.org/issues/57/9.html

[23] ASLR on the Line
http://www.cs.vu.nl//-herbertb/download/papers/anc_ndss17.pdf