Defence Against Code Injection Attacks

Hussein Alnabulsi[1], Quazi Mamun[1], Rafiqul Islam[1], and Morshed U. Chowdhury[2(✉)]

[1] School of Computing and Mathematics, Charles Sturt University, Albury, Australia
{halnabulsi,qmamun,mislam}@csu.edu.au
[2] School of Information Technology, Deakin University, Melbourne, Australia
muc@deakin.edu.au

Abstract. Code injection attacks are considered serious threats to the Internet users. In this type of attack the attacker injects malicious codes in the user programs to change or divert the execution flows. In this paper we explore the contemporary defence strategies against code injection attacks (CIAs) and underline their limitations. To overcome these limitations, we suggest a number of countermeasure mechanisms for protecting from CIAs. Our key idea relies on the multiplexing technique to preserve the exact return code to ensure the integrity of program execution trace of shell code. This technique also maintains a FIFO (first in first out) queue to defeat the conflict state when multiple caller method makes a call simultaneously. Finally, our technique can provide better performance, in terms of protection and speed, in some point compared to the CFI (control flow integrity) as well as CPM (code pointer masking) techniques.

Keywords: Security · Code injection attacks · Malicious

1 Introduction

Code injection attack is a malicious activity where a malware code placed by hacker in the memory of a system either to cause damage on the system, spy on the user, or to steal user's information. An attacker tries to take the control of the program flow using code injection attacks by changing the return address of shell code. An example of code injection attacks is the stack-based buffer overflow, which overwrites the return address. In addition, more advanced techniques are exist such as indirect pointer overwrites, and heap-based buffer overflows, where a code pointer can be overwritten to divert the execution control to the attacker's shell code. According to National Institute of Standards and Technology (NIST 2013) [1], code injection attack represents a high priority of all types of vulnerabilities and that must be mitigated. According to (NIST) (9.86 % of attacks is buffer over flow) came after SQL injection attacks and XSS attacks (16.54 % of attack, 14.37 % of attack) respectively [1].

There are many countermeasures techniques are available against code injection attacks but each has limitations. Below, we describe them briefly:

1. StackGuard: In this technique a canary secret value is placed between the buffer and the return address. Thus when an intruder tries to change the return address in the

J. Tian et al. (Eds.): SecureComm 2014, Part II, LNICST 153, pp. 237–251, 2015.
DOI: 10.1007/978-3-319-23802-9_19

victim program StackGuard detects these changes. As the canary value is random 32-bit wide then it is very hard for an attacker to guess the value of the canary. The canary value is usually chosen at the time when program starts. There are four types of canaries that have been used: Random Canary, Random XOR Canary, Null Canary, Terminator Canary [9, 11, 14].
The limitation of StackGuard is that this technique does not work well when indirect pointer overwrite attacks occurred. In this type of attack the attacker overwrites an unprotected pointer and then inserts a value in the stack. Whenever, the application changes the pointer and overwrites the value with the integer, the attacker writes a random value in any location in the memory by manipulating the pointer and the integer. Thus, the attacker can write any value over the return address of the stack. Moreover, StackGuard technique is incompatible with Linux kernel [4, 8, 11].

2. Address Space Layout Randomization (ASLR): This technique is used to protect the system from buffer over flow attacks. It works by randomizing the base address of structures such as the heap, stack, and libraries, making it hard for an attacker to find injected shell code in memory of the system. Also if an attacker succeeds in overwriting a code pointer, the attacker does not know where to point it [10]. Windows Vista provides Address Space Layout Randomization as a basis on a per image, so any executable image which have a PE header, such as (.dll or .exe) can be used in Address Space Layout Randomization (ASLR) [13].
   Limitation of the Address Space Layout Randomization countermeasure is that an attacker may use a new technique called heap-spraying, in this technique an attacker fills the heaps with many copies of the malicious shell code, and then jumps any location in the memory, this operation gives a good probability to an attacker that he will access to his shell code in memory [4]. However Address Space Layout Randomization is not effective against exploit code for single flaw, and for brute force attack [12].
3. Memory Management Unit Access Control Lists (MMU ACLs): it allows applications to mark memory pages as non-executable memory, and supported by Control Processing Units (CPUs). In this technique, memory semantic consists of three components: the first component separates readable and writable pages, the second component makes stack, heap, anonymous mapping, and mark memory pages as non-executable memory, the third one is enforcing ACL (Access Control List, which is a set of data that informs the operating system (OS) about permissions, access rights, and privileges (read, write, and execute) for users in the computer system object [15]), which control the operation of converting the non-executable memory to executable memory and vice versa or denying the conversion [1, 14].
   However, the MMU ACLs are not supported by some processors, as the technique breaks applications which expect the heap or stack to be executable. It is possible that an attacker injects a crafted fake stack, then an application will unwind the fake stack instead of the original calling stack, the attacker then directs the processor to arbitrary function in program code or library, and it is also possible that the attacker marks the memory where he injected his shell code as executable and jumping to it [1].
4. StackShield: it is developed from StackGuard to protect against stack smashing attacks, and exploitation of stack based buffer overflows. This technique applies a random

secret canary value. This technique is comparatively better, however it cannot protect against indirect pointer overwriting attacks [8, 17].

5. Control Flow Integrity (CFI): This technique provides a good guarantee against code injection attacks, it determines control flow graph for every program, and to each control flow destination of a control flow transfer it gives a unique ID. The CFI will know if there are code injection attacks by comparing the destination ID with expected ID before transferring the control flow to destination, if they are not equal, the application will kill. Otherwise the program will proceed as normal.
   In comparison with other countermeasures CFI is considered very slow [2, 4].
6. CPM (masking code pointers) is a countermeasure against code injection attacks, CPM does not detect memory corruption or prevent overwriting code pointer, but it is hard for an attacker to make a successful code injection attack. It provides a mask to every return addresses and function pointer of the program and masks the global offset table, the masking operation relies on logic operation such as OR, AND operations to prevent code injection in the memory of the system [1]. The limitation of CPM (masking code pointers) is that it does not give a good guarantee of protection against code injection attacks.

Our key idea relies on the multiplexing technique to preserve the exact return code to ensure the integrity of program execution trace of shell code. This technique also maintains a FIFO (first in first out) queue to defeat the conflict state when multiple caller method makes a call simultaneously.

To alleviate the aforementioned problems with the existing countermeasures, we propose a technique which relies on the multiplexer idea. To preserve the exact return code to ensure the integrity of program execution trace of shell code. As each method has a particular ID, using the multiplexer method an attacker will not be able to divert the return address to the attacker's shell code. This technique also maintains a FIFO (first in first out) queue to defeat the conflict state when multiple caller method makes a call simultaneously. The proposed technique has some similarity with the CFI; however it improves the main problem of the CFI, which is being slow.

The rest of paper is organized as follows: The next section provides related work. Section 3 illustrates the idea of our approach. Section 4 provides complexity analysis between CFI methodology and multiplexer methodology, and between CPM methodology and Counter methodology. Finally, the paper is concluded in Sect. 5.

2 Related Work

According to [3] the authors presented a design of MoCFI (Mobile CFI) and implemented a framework of Smartphone platforms, the standard platform that they focused is ARM architecture. The ARM architecture is standard platform of smartphones. It prevents control flow attacks by using control flow integrity (CFI) method. In the paper authors showed the enforcement of CFI that applied on the ARM platform, and implemented CFI framework for Apple iOS. The result showed that the application mitigates Control Flow Attacks. The limitation of MoCFI is that it can only applicable in mobile phone applications.

In [4] authors presented a CPM countermeasure against code injection attacks which does not depend on secret value (stack canaries), but relies on masking the return address

functions, masking function pointer, and masking global offset table. However it gave a good protection against code injection attacks but there are some scope for an attacker to success his attack, therefore it does not give a full protection against code injection attacks.

Lee *et al.* [5] presented secure return address stack (SRAS) which is hardware-based to prevent code injection attacks by verifying modification of return address. To apply this methodology needs low cost modification of the processor and operating system (OS), the hardware protection can be applied to executable code and new program. The impact of performance of the applications using this hardware-based is negligible, it does not impact on performance of return instructions and procedure call. However SRAS requires a hardware modification that some time hard to apply because of compatibility with the processor and operating system (OS) of the system.

Zhang *et al.* [6] presented a new protection method called Compact Control Flow Integrity and Randomization (CCFIR), to solve the limitation of CFI by collecting all targets of indirect control transfers instructions in a section called "Springboard" in a random order. By using "Springboard" section CCFIR will ensure indirect transfers easier than CFI. Result showed that CCFIR eliminate control flow attacks such as return-into-libc and ROP, but this technique still use the same approach of CFI.

In [7] Xia *et al.* presented a system called CFIMon, which is the first system can detect a control flow attacks without any changes to applications (binary code or source) and it does not require any special hardware. It works by collecting legal control flow transfers and uses branch tracing store mechanism to analyse runtime traces, and detects the code injection attacks. The CFIMon uses two phases: offline phase which builds a set of target addresses for every branch instruction, and online phase that applied a number of rules to diagnoses possible attacks. The limitation of CFIMon is that it gives a false alarm (false positive or false negative) when detecting code injection attacks.

The motivation of our work is to improve CFI speed by applying a multiplexer to divert every return addresses to its address memory locations, and overcome problem with CPM which did not give a complete guarantee to protect against code injection attacks by using a counter of one's in return addresses. Our approaches addresses these issues and have a significant countermeasures to protect against code injection attacks.

3 Our Proposed Approaches

In this section, we propose two different approaches to protect the code injection attacks. The first approach is a *multiplexer application approach*, which can protect the return address of each legitimate method, residing in a non-writable memory location. In our second approach we apply a matrix to keep the number of 1s of the return addresses of each legitimate method, and later we compare it with the calling method for similarity testing. The following subsections describe our approaches in details.

Multiplexer approach:

In this approach we apply a software module called *MUX_App* and every method in the system must call the *MUX_App* to get its return address. Each legitimate method is provided with a unique ID and will have its own return address. Our assumption is to

keep *MUX_App* in a memory location which is non-writable. As a result the *MUX_App* remains protected from the attackers to divert the program execution (Fig. 1).

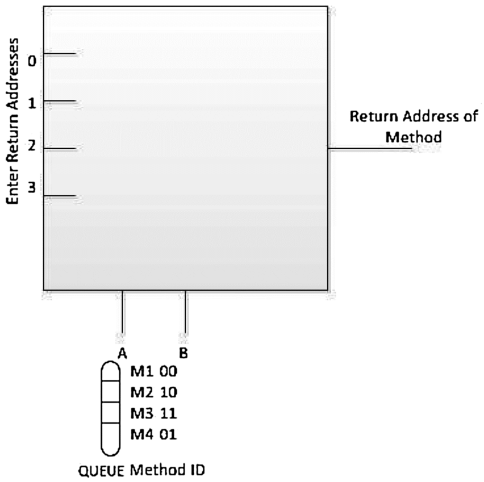

Fig. 1. Multiplexer with the return address of the method

We provided a queue (first-in first-out) to the multiplexer application to manage calling operations to prevent multiple calls for the multiplexer application in the same time. Therefore it will give a protection from code injection attacks and will improve speed significantly compare to CFI technique.

Figure 2 shows the assembly code for the CFI method. Note that, the code contains a comparison operation (cmp [ecx], 12345678h), which make the CFI method slower compared to the proposed multiplexer approach.

Source			destination		
Bytes (opcodes)	x86 assembly code	Comment	Bytes (opcodes)	x86 assembly code	Comment
FF E1	jmp ecx	; a computed jump instruction	8B 44 24 04	mov eax, [esp+4]	; first instruction of destination code
81 39 78 56 34 12	cmp [ecx], 12345678h	; compare data at destination	78 56 34 12	DD 12345678h	; label ID, as data
75 13	jne error_label	;if not ID value, then fail	8B 44 24 04	mov eax, [esp+4]	; destination instruction
8D 49 04 FF E1	lea ecx, [ecx+4] jmp ecx	; skip ID data at destination ; jump to destination code			

Fig. 2. CFI low-level language code [2]

The policy of our methodology is that the software execution must follow a path of Control Flow Graph (CFG) which determined ahead of time as shown in Fig. 3. The definition of Control Flow Graph (CFG) is a representation using graph notation, of all paths that might be traversed through a program during its execution. The CFG can be defined by analysing source code, or execution profile [2, 16]. Our methodology based on CFG; its enforcement can be implemented in software as shown in Fig. 3. It is very close to CFI in execution, it works by preventing an attacker divert code execution, by diverting to a static return addresses using a multiplexer for every return addresses and every static application program. Therefore attackers cannot divert the code execution to their shell code as shown in Fig. 4. The figure shows that an attacker tried to divert the execution method M1 by calling it from an attacker's method M2, but when method M1 calling the multiplexer to get its return address then an attacker will fail because the multiplexer will give the right return address of M1 to method M1.

The difference between our methodology with CFI, is that our methodology is faster because it diverts the return address methods immediately by using a multiplexer's application that contains a static return addresses for every application's methods, but the CFI is working by comparing the ID of the destination with expected ID, and if they are not equal, the program will be killed, but if they are equal the program proceeds as normal as shown in Fig. 2. The comparing operation takes a longer time than our methodology [1].

Figure 5 illustrates how the attacker can successfully inject shell code in process.

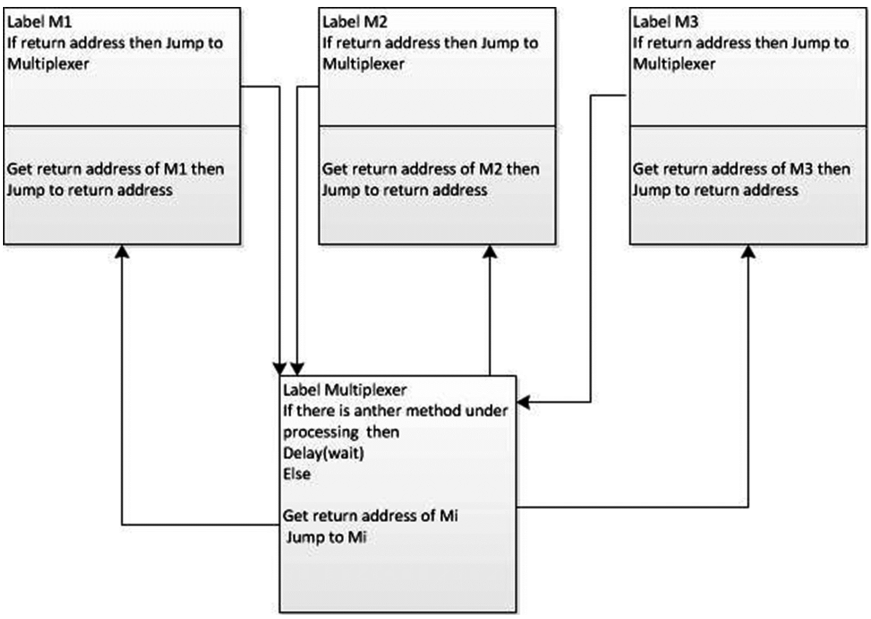

Fig. 3. CFG control flow graph

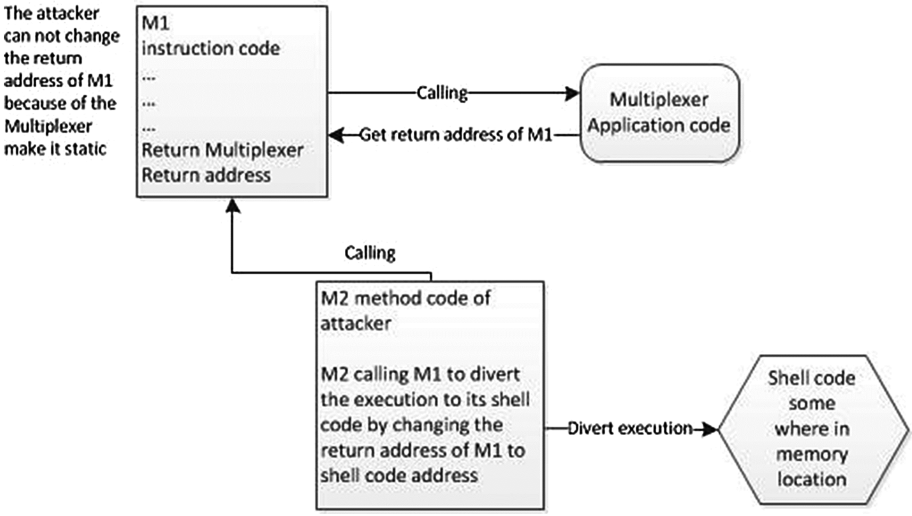

Fig. 4. An attacker tries to divert the execution to his shell code

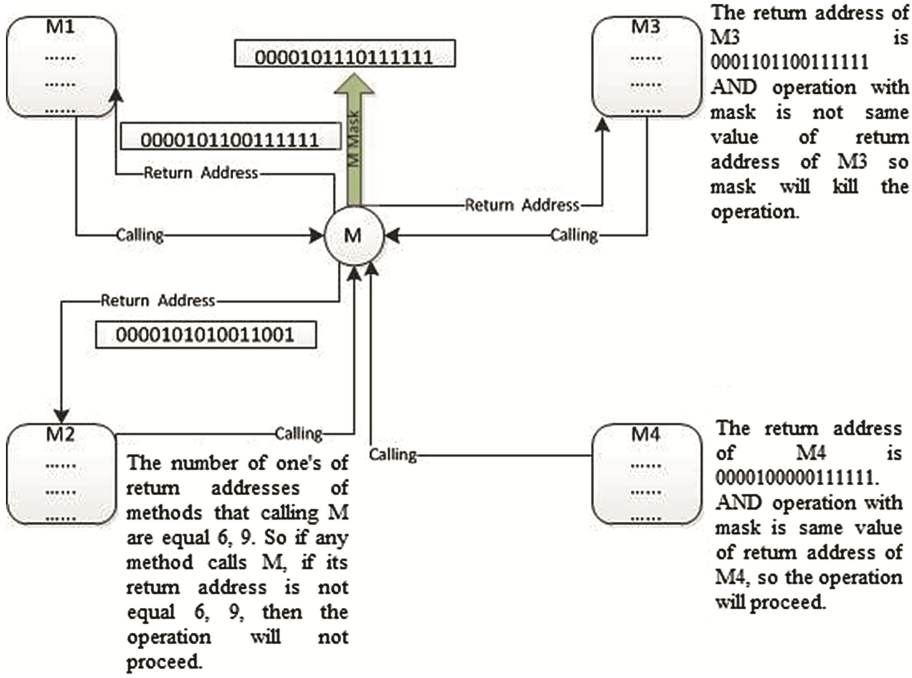

Fig. 5. Explaining for example 1

Example 1:

Consider two caller methods M1 and M2 and a call method M. During the compilation the system knows the return addresses of M say 0 × 0B3F (0000101100111111) for method M1, and 0 × 0A99 (0000101010011001) for method M2. The mask can be calculated by performing bitwise logical OR operations of the two return addresses. Therefore, the final mask is calculated as 0 × 0BBF (0000101110111111). As a result, bitwise logical AND operation between the final mask and a return address produces the same return address. On the other hand, an invalid return address does not produce the same return address after AND operation with the final mask. This is a basic technique to identify incorrect/malicious return address, as proposed in [1].

However, this technique fails to protect the system all the time. For instance, assume a malicious method M3 calls the method M. Assume the return address of M3 is 0001101100111111. In this case, the bitwise logical AND operation 0000101110111111 && 0001101100111111 does not produce 0001101100111111. Therefore, the method M3 will be detected as malicious method. On the other hand, consider another malicious method M4 for which the return address is 0000100000111111. Performing a logical bitwise AND operation between the final mask and the return address does produce the correct return address 0000100000111111&&0000101110111111 = 0000100000111111. Therefore, the method M4 increases the false negative instance.

Counter Matrix:

To improve the possibility to identify a malicious method, our proposed solution uses the same masking technique [1] after an additional phase, called *Counter Matrix,* of counting 1s of the return addresses. In this bit counting "process numbers of one's (1's) are counted and compare with that of the return address as shown in the following algorithm.

```
if (bit_count())==0  // compare the number of ones
      return False; // Malicious method identified
else run CPM(); // the system will consider the    //
                method as vulnerable and will
                // call CPM process.
```

For example, the return addresses of M1, M2 have 6 and 10 one-bits respectively. Therefore M4, M5 could not call M3 because the number of one's is 7, 3 respectively and the operation process will kill.

After counting operation if the code injection attack is not detected, then a mask operation will be done by using OR operation with return addresses methods that is M3, then AND operation will conduct with every return address methods of M3, therefore our method will improve the time. Because it - kills the operation if the counting operation detect differences in count of one's for each method, and it will ensure security. The security is enhanced because the masking operation (CPM) is conducted after counting operation. However if still there is a small space for the attacker to pass his attack then it will fail because we use two techniques together (Counter of One's technique and CPM

technique). Therefore, it is not needed more time in execution if the first technique (Counter of One's technique) detects an attack operation it will kill the operation of method without executing second technique (CPM technique). Here we preview example 2 with its Fig. 6.

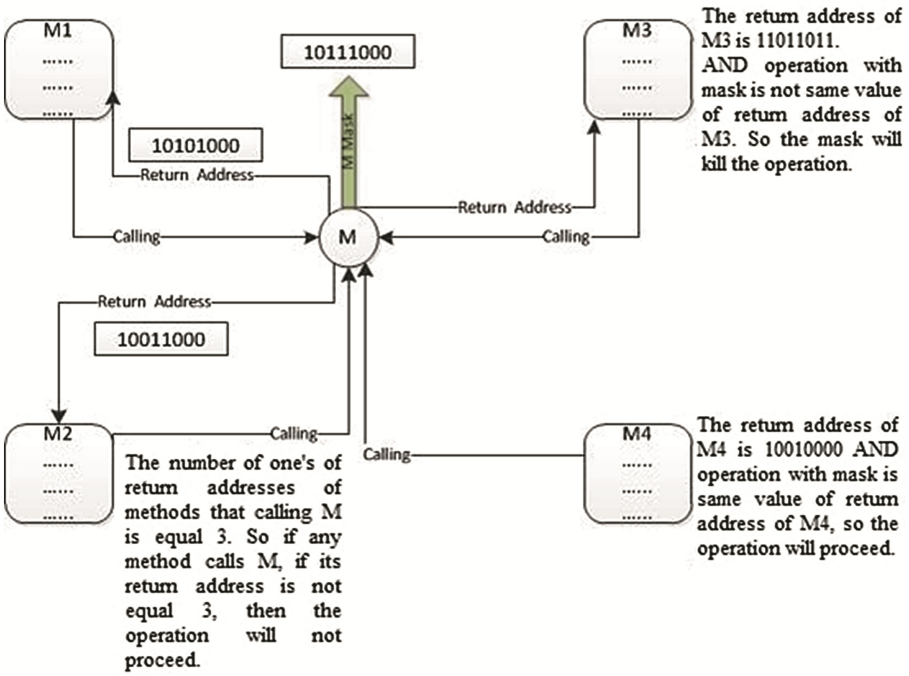

Fig. 6. Explaining for example 2

Consider a method M which is being called by two other methods M1 and M2. The mask of M is calculated by performing logical bitwise OR operation of the return addresses of M1 and M2.

Thus, Masking M = 10101000 OR 10011000 = 10111000.

We consider a method M3, which contains malicious code and this method wants to call the M.

Therefore when AND operation conducted with the return address of M3 (11011011) with the making code 10111000, we will not get the same return address value of M3 (11011011), thus the attacker cannot continue his attacks, and he cannot divert the return address of method M into his shell code in memory and cannot execute it because the application will kill. In the same Fig. 6 illustrate a preview example of masking failure, Consider M4 as a malicious code that want to call the method M.

The AND operation of the return address of M4 with the mask of M, the result will be the same value of the return address of M4, so the application will continue.

Therefore if we apply our technique in this problem it will count the numbers of 1's for every return addresses that has called method M. In this example its 3 (1's), and the return address of M4 is 4, and for M3 is 6.

By applying this technique the application will kill, because the number of (1's) of M3 and M4 are not equal with 3 (1's).

Figure 7 shows our propose algorithm of multiplexer methodology with its time execution.

```
If call method  => O(1)
Going to Multiplexer location  => O(1)
Give ID of method to Multiplexer  => O(1)
Get return address for method ID  => O(1)
Jump to return address location => O(1)
```

Fig. 7. Algorithm of multiplexer methodology with its time execution

The total execution time for multiplexer methodology is equal $O(1) + O(1) + O(1) + O(1) + O(1) => O(1)$ as shown in Fig. 7. Our algorithm of counter methodology with its time execution is illustrated in Fig. 8.

```
If call method   => O(1)
Going to counter location  => O(1)
Compare count of one's  => O(n)
 If true  = > O(1)
   Go to mask location  => O(1)
   AND Operation  => O(n)
     If true  => O(1)
        Go to return address location => O(1)
     else
        error kills operation  => O(1)
 else
     error kills operation  => O(1)
```

Fig. 8. Algorithm of counter methodology with its time execution

The execution time for counter methodology according to Fig. 8 is $O(N) + O(N) + O(1) + O(1) + \ldots. + O(1) = O(2\,N)$. Figure 9 shows an assembly code for counter methodology.

```
for (int i=0; i<a.length; i++)
051E6B40: 04      iconst_0              //get 0                 : A
051E6B41: 3E      istore_3              //  store it in i
051E6B42: A70011  goto      0x051E6B53  //go to test i<a.length near bottom

          if (a[i] = n)
051E6B43: 2B      aload_1               //get a's base address : B
051E6B44: 1D      iload_3               //  get i
051E6B45: 2E      iaload                //  get a[i]
051E6B46: 1C      nload_2               //get n
051E6B47: A70011  goto      0x051E6B52  //go to increment C
051E6B49: A40007  if_icmple  0x051E6B50 //go to test near bottom if <=

051E6B50: 840301  iinc                  //increment i          : C

     Count=Count+1
051E6B51: 1E      Cload_4               //get count             : D
051E6B52: 840303  Cinc                  //increment C           : E
051E6B53: 1D      iload_3               //get i                 : F
051E6B54: 2B      aload_1               //get a's
051E6B55: BE      arraylength           // length
051E6B56: A1FFED  if_icmplt  0x051E6B43 //go to if above if <
Compare Count with count of one's in method
051E6B57: 2E      Cload_5               //get count from method   : G
          if (b[s] = C)
051E6B58: 2B      bload_1               //get b base address : H
051E6B59: 1D      sload_1               //  get s
051E6B60: 2E      sbload                //  get b[s]
051E6B61: 1C      cload_3               //get c
Then same operations of mask methodology
```

Fig. 9. An assembly code for counter methodology [19]

Figure 9 also shows the implementation of the countermeasure technique, which requires 21 assembly instructions. Although the implementation of the proposed technique requires few extra instructions, the proposed technique detects more anomalies than the CPM technique. This is because the proposed technique uses parity checking in addition to the masking technique.

Figure 10 shows use of C code for quicksort algorithm to illustrate our multiplexer methodology; we preview the following C function, this function code return the median value of an array of integers [18].

```
int median (int* data, int len, void* cmp )
{ int tmp [MAX_LEN];
Memcpy ( tmp, data, len*sizeof(int) );
Qsort ( tmp, len, sizeof (int), cmp );
Return tmp [len/2];
}
```

Fig. 10. Quicksort algorithm [18]

The time complexity of quicksort algorithm is O (N*log N) [20]. In Figs. 11, 12 and 13 show assembly codes of quicksort algorithm, quicksort with CFI, and quicksort with our multiplexer technique respectively.

```
Regular_qsort:
...
push  ebx
mov  eax, esi
call  shortsort
add  esp, 0ch
...
push  edi   ;  an attack is
push  edx  ;  possible by
call  [esp+comp_fp]  ;   Going to X
add  esp,  8
test  eax,  eax
jle  lable_lessthan


Regular_library_function:
mov  edi, edi
push  ebx
mov  ebx, esp
push  ecx
...
pop  ebp
X:  mov  esp, ebx
pop  ebx
ret
```

Fig. 11. Quicksort assembly code [18]

The Qsort algorithm code is vulnerable as shown in Fig. 11. Figure 12 shows the assembly code of Qsort with CFI. If an attacker overwrites the comparison function cmp before it is passed to Qsort, an attacker can exploits this point to divert the execution to his shell code when the Qsort method calls the corrupted comparison function cmp. This has been labelled as X in Figs. 11, 12, and 13.

The Qsort with CFI assembly code in Fig. 12 includes ID checks before call instructions, therefore CFI methodology will prevent the exploiting Qsort code from an attacker, because of the ID checks will happen before call instructions and this prevents any exploitation in calling instructions in the code.

In Fig. 13 shows Qsort with multiplexer methodology that will protect from an attacker exploiting because when call function has happened, an attacker has changed the return address of method. Therefore, next instruction is to get the return address from a multiplexer (prefetchnta [AABCCDEEh]: line 5 in Fig. 13) which means to go to this location in memory to get the return address of the method. As a result incorporating multiplexer technique is able to protect from attacking such as CFI methodology, but also improves execution time [18].

```
Qsort_with_CFI
…
push ebx
mov eax, esi
call shortsort
prefetchanta [AABCDDEEh]   ; tacking data from this location
add esp, 0Ch
…
push edi
push ebx
mov eax, [esp+comp_fp]
cmp [eax+4], 12345678h      ; CFI check
jne error lable              ; Prevents
call eax                     ; going to X
prefetchanta [AABCDDEEh]  ; tacking data from this location
add esp, 8
test eax, eax
jle label_lessthan
…
```

Fig. 12. Quicksort assembly code with CFI [18]

```
Qsort_with_Our_ methodology
…
push ebx
mov eax, esi
call shortsort
prefetchanta [AABCCDEEh]   ; tacking data from this location
add esp, 0Ch
…
push edi
push ebx
mov eax, [esp+comp_fp]
call eax                     ; going to X
prefetchanta [AABCCDEEh]  ; tacking data from this location
add esp, 8
…
```

Fig. 13. Quicksort assembly code with multiplexer methodology [18]

4 Complexity Analysis Between CFI, Multiplexer Methodologies and CPM, Countermeasure Methodologies

It also shows in Figs. 12 and 13 that our multiplexer methodology requires only 11 lines of code instead of 12 lines of code used in CFI for implementation, which is an evidence of improving execution time. Table 1 shows a complexity analysis between our method, CFI and CPM. Our method gives better results compare to other methods.

Table 1. Complexity analysis between CFI, Multiplexer methodologies and CPM, Countermeasure methodologies

	Execution time	Security	Lines of code
CFI	O(n)	No vulnerability	15 lines of code* (N time)
Multiplexer	O(1)	No vulnerability	11 lines of code* (N time)
CPM	NA	Good protection	NA
Counter	O(2n), or O(2n) + CPM execution time	It uses 2 approaches together (Countermeasure + CPM)	21 + CPM's line of code

5 Conclusion

In this paper we propose two approaches to defence against code injection attacks. One of the approaches augments '1' bit counting technique to modify the Masking Code Pointer (CPM) [1]. The augmentation technique improves the probability of identifying malicious code compared to CPM. The proposed technique provides more protection without introducing time complexity.

The second methodology relies on the multiplexer technique, which is based on the Control Flow Integrity (CFI) however performs faster than the CFI technique. The time complexity of the proposed multiplexer technique is O(n) compared to the time complexity of the CFI which is O($2n$).

In our future work, we will apply our counter technique and multiplexer technique using a Linux environment and SPEC CPU2006 Integer benchmarks.

References

1. Philippaerts, P., Younan, Y., Muylle, S., Piessens, F., Lachmund, S., Walter, T.: CPM: masking code pointers to prevent code injection attacks. ACM Trans. Inf. Syst. Secur. (TISSEC) **16**(1), Article No. 1 (2013)
2. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control flow integrity principles, implementations, and applications. ACM J. **13**, 4 (2006)
3. Davi, L., Dmitrienko, A., Egele, M., Fischer, T., Holz, T., Hund, R., Nurnberger, S., Sadeghi, A.: MoCFI : a framework to mitigate control-flow attacks on smartphones. IETF J. **4**, 32–44 (2012)

4. Philippaerts, P., Younan, Y., Muylle, S., Piessens, F., Lachmund, S., Walter, T.: Code pointer masking: hardening applications against code injection attacks. In: Holz, T., Bos, H. (eds.) DIMVA 2011. LNCS, vol. 6739, pp. 194–213. Springer, Heidelberg (2011)
5. Lee, R.B., Karig, D.K., McGregor, J.P., Shi, Z.: Enlisting hardware architecture to thwart malicious code injection. In: International Conference on Security in Pervasive Computing (SPC 2003), pp. 237–252, Boppard, Germany (March 2003)
6. Zhang, C., Wei1, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W.: Practical control flow integrity and randomization for binary executables. In: 34th IEEE Symposium on Security and Privacy (Oakland), San Francisco (May 2013)
7. Xia, Y., Liu, Y., Chen, H., Zang, B.: CFIMon: detecting violation of control flow integrity using performance counters. In: 42nd Annual IEEE/IFIP International Conference, pp. 1–12 (2012)
8. Richarte, G.: Four different tricks to bypass StackShield and StackGuard protection. J. Comput. Virol. **7**(3), 173–188 (2002)
9. Etoh, H., Yoda, K.: Protecting from stack-smashing attacks. IBM Research Division, Tokyo Research Laboratory (June 2000)
10. Bhatkar, S., DuVarney, D.C., Sekar, R.: Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In: 12th USENIX Security Symposium, USENIX Association (2003)
11. Cowan, C., Beattie, S., Day, R.F., Pu, C., Wagle, P., Walthinsen, E.: Protecting systems from stack smashing attacks with StackGuard (May 2005)
12. Shacham, H., Page, M., Pfaff, B., Goh, E., Modadugu, N., Boneh, D.: On the effective of address-space randomization. In: CCS 2004 Proceedings of the 11th ACM Conference on Computer and Communications Security, pp. 298–307 (October 2004)
13. Whitehoue, O.: An analysis of address space layout randomization on Windows Vista. Symantec Adv. Threat Res. (February 2007)
14. Silberman, P., Johnson, R.: A Comparison of Buffer Overflow Prevention Implementations and Weaknesses. iDEFENSE Inc., Dallas (2004)
15. ACL (2014). http://www.webopedia.com/TERM/A/ACL.html
16. Control flow graph (April 2014). http://en.wikipedia.org/wiki/Control_flow_graph
17. Youna, Y., Pozza, D., Piessens, F., Joosen, W.: Extended Protection Against Stack Smashing Attacks Without Performance Loss, pp. 194–213. Springer, Berlin (2006)
18. Abadi, M., Budiu, M., Erlingsson, U., Ligatti, J.: Control flow integrity principles, implementations, and applications. ACM J. **13**(1), Article 4 (2009)
19. Pattis, R.E.: https://www.cs.cmu.edu/afs/cs/Web/People/pattis/15-1XX/15-200/lectures/aa/index.html. Accessed June 2014
20. How to find time complexity of an algorithm. http://stackoverflow.com/questions/11032015/how-to-find-time-complexity-of-an-algorithm