FRProtector: Defeating Control Flow Hijacking Through Function-Level Randomization and Transfer Protection

Jianming Fu[1,2,3], Rui Jin[1,2(✉)], and Yan Lin[4]

[1] Computer School, Wuhan University, Wuhan, China
jmfu@whu.edu.cn, r-jin@foxmail.com
[2] State Key Laboratory of Aerospace Information Security and Trusted Computing of the Ministry of Education, Wuhan, China
[3] State Key Laboratory of Software Engineering, Wuhan University, Wuhan, China
[4] School of Information Systems, Singapore Management University, Singapore, Singapore
yanlin.2016@phdis.smu.edu.sg

Abstract. Return-oriented programming (ROP) and jump-oriented programming (JOP) are two most common control-flow hijacking attacks. Existing defenses, such as address space layout randomization (ASLR) and control flow integrity (CFI) either are bypassed by information leakage or result in high runtime overhead. In this paper, we propose *FRProtector*, an effective way to mitigate these two control-flow hijacking attacks. *FRProtector* shuffles the functions of a given program and ensures each function is executed from the entry block by comparing the unique label for it at *ret* and indirect *jmp*. The unique label is generated by XORing the stack frame with return address instead of with a random value and it is saved in a register rather than on the stack. We implement *FRProtector* on LLVM 3.9 and perform extensive experiments to show *FRProtector* only adds on average 2% runtime overhead and 2.2% space overhead on SPEC CPU2006 benchmark programs. Our security analysis on RIPE benchmark confirms that *FRProtector* is effective in defending control-flow hijacking attacks.

Keywords: Control flow hijacking · Control flow protection
Function-level randomization · Code reuse attack

1 Introduction

Control-flow hijacking [1] is one of the most common attack method today, which modifies the target of control flow transfer instruction (e.g., indirect jump, function return instruction) to the code carefully crafted by the attacker. The traditional control-flow hijacking [1], code injection attack, redirects the control flow to the code snippet (shellcode) which is injected by the attacker through memory corruption vulnerabilities. This attack has been defeated by data execution prevention (DEP) [2]. Today, attackers are widely using code reuse attacks

X. Lin et al. (Eds.): SecureComm 2017, LNICST 238, pp. 659–672, 2018.
https://doi.org/10.1007/978-3-319-78813-5_34

(CRA) [3,4], which re-construct code snippets (gadgets) that already exists in code segments to achieve the malicious purpose.

To counter control-flow hijacking, several hardening techniques have been widely adopted, including stack guard (GS) [5], address space layout randomization (ASLR) [6] and control flow integrity (CFI) [7–11]. Stack guard inserts an unpredictable number between return address and local variables. This unpredictable number is obtained by XORing a random value with the stack frame value. ASLR increases the entropy of the process by randomizing the base address of the memory segment. CFI constructs the control flow graph (CFG) of the program statically and forces the program to comply with the rules of the CFG. It marks the valid targets of indirect control flow transfers with unique labels. Before each transfer instruction of the program, CFI checks whether the label of the destination address is the same as expected.

However, the first two can be bypassed by BlindROP [12] or information disclosure [13], and the last one usually brings expensive runtime overhead. Moreover, having an accuracy static analysis is known to be hard. In this paper, we present *FRProtector*, a more effective way to counter control-flow hijacking. The purpose of *FRProtector* is to make it hard for attackers to guess the expected code location and to ensure each function is executed from the entry block. *FRProtector* first reorders the locations of functions. But this function-level randomization is a mitigating method that can not provide deterministic defense. Sometimes, it can be bypassed by well-structured information disclosure. In order to obtain better security, for each function, a unique label is generated by XORing the stack frame pointer with the return address. Then, *FRProtector* adds runtime checks into the program to check whether the label generated at the ret and indirect jump instruction is the same one.

FRProtector is similar to GS in some respects, but it achieves better security. First, *FRProtector* can effectively detect attacks that do not leverage stack buffer overflow to overwrite the return address as it uses return address to generate the label, while GS cannot defend this kind of attack. In addition, *FRProtector* stores the label in the register rather than storing it onto the stack, which increases the difficulty for the attacker to obtain it. Finally, *FRProtector* also checks the label before indirect jump instructions to defend control-flow hijacking attacks that modify the registers used in indirect jump instructions.

Although the idea sounds simple, the key to a successful defense that can gain acceptance by developers is a low runtime overhead in the resulting binary executable. To achieve this goal, we have implemented *FRProtector* on LLVM 3.9. The extensive experiments show that *FRProtector* results in a small runtime overhead of 2% and space overhead of 2.2% on average. *FRProtector* is effective as it prevents all attacks that overwrite the return address in the RIPE benchmark [14].

In summary, this paper makes the following contributions:

1. We propose *FRProtector*, an effective control-flow hijacking defense that reorders the locations of functions and ensures a function is executed from

the entry block by comparing the unique label for it at ret and indirect jump instruction.
2. We perform extensive experiments to show *FRProtector* results in low runtime and space overhead.
3. We compare *FRProtector* with CFI and GS. *FRProtector* achieves similar security compared to CFI. While comparing with GS, *FRProtector* provides better security.

The paper is organized as follows. Section 2 introduces the background of control-flow hijacking attack and the thread model. Section 3 presents the *FRProtector*. Section 4 describes the implementation of our solution on LLVM. We demonstrate the efficiency of *FRProtector* with extensive performance evaluations and present the security analysis in Sect. 5. Section 6 compares *FRProtector* with CFI and Stack Guard. Related work and conclusion are presented in Sects. 7 and 8.

2 Background

In this section, we start with a brief summary of control-flow hijacking and then define our threat model.

2.1 Control Flow Hijacking

Control flow hijacking is a kind of memory corruption attack. The attacker redirects the program's code pointer to the location of the shellcode or gadgets. Shellcode is used for the code injection attack, while gadgets for CRA.

Code Injection Attack. The attacker can inject malicious code into the memory, and then redirect the control flow to the memory address of malicious code through memory vulnerabilities. For example, an attacker controls the area near the overflow area through a stack overflow vulnerability, and injects malicious code into this area, then modifies the return address to the first instruction of the malicious code. Now, it can be defeated by Data Execution Prevention (DEP).

Code Reuse Attack. Code Reuse Attacks (CRA) use code in the program or libraries to construct code snippets (called gadgets), each of which has a specific feature (e.g., writing a specified value to a fixed register). A gadget is a small sequence of binary code that ends in an indirect instruction. By chaining different functional gadgets, an attacker can construct a code execution sequence that implements the same functionality as malicious code. For example, by constructing the appropriate parameters, the return-into-libc attack, a kind of CRA, redirects the control flow to the standard libraries to call a library function. At this stage the most popular code reuse attacks are Return-Oriented Programming (ROP) [3] and Jump-Oriented Programming (JOP) [4].

ROP is an exploit technique that has evolved from stack-based buffer overflows. In ROP exploits, gadgets end in ret instructions. By carefully crafting a sequence of addresses on the software stack, an attacker can manipulate the ret instruction to jump to arbitrary addresses that corresponding to the beginning of gadgets. ROP has proved to be Turing complete. JOP is similar as ROP, excepting that JOP uses the indirect jump instructions to modify the program's control flow.

To complete the CRA, one of the challenges is to identify the exact address of each gadget in the memory space. ASLR makes the attacker harder to get these accurate addresses by randomizing the base address of the target program. But information disclosure [13] or brute force can assist the attacker to find the accurate address. For instance, Just-In-Time Code Reuse [15] uses memory disclosure to bypass ASLR, and Blind-ROP [12] uses brute force.

2.2 Threat Model

The proposed defense, *FRProtector*, is aimed to protect a vulnerable application against control-flow hijacking attacks, including ROP and JOP attacks. The left of Fig. 1 shows an example of ROP attack. Function 1 has the input instruction, and the attacker uses it to push the payload into the stack. Function 2 has an overflow vulnerability, in which the attacker can modify the return address to the payload. So, if Function 1 is called before Function 2, the attacker will hijack the control flow successfully. An example of modifying the stack frame to construct CRA is shown in the right of Fig. 1. Function 3 is similar to Function 1. Function 4 has a vulnerability which modifies the register of stack frame to the payload. So, when Function 4 is called, the control flow will be transferred to the address where the payload pointing to.

It seems that Stack Guard (GS) can counter both attacks mentioned above. But if the attacker just overwrites the return address without the overflow vulnerability, GS cannot defend it. Moreover, such as $func5$ in Fig. 1, attackers can modify the registers that are used in indirect jump instructions to point to the location where the control flow will be transferred to. It is out of the range that GS can protect. These four kinds of exploitations need to be considered. We can divide them into three categories.

- The return address is redirected to the gadget address. It can be achieved by stack buffer overflow or by modifying the return address directly.
- The stack frame is modified to the address where the payload (gadgets chain) located in.
- The register used in the indirect jump instruction is modified to the gadget address.

On the other hand, we assume attackers cannot modify the code segment, because the corresponding pages are marked read-executable and not writable. This assumption ensures the integrity of the original program code instrumented at compile time. Meanwhile, the attacker cannot examine the memory dump of the running process and is unaware of how exactly the code is randomized. Our assumptions are consistent with prior work in this area.

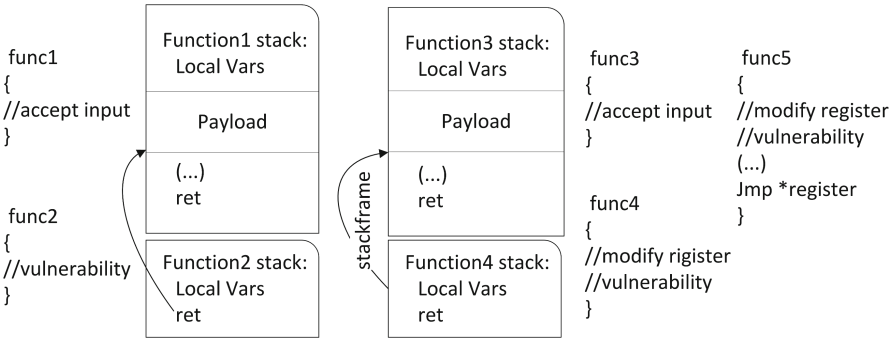


Fig. 1. Control-flow hijacking

3 FRProtector Design

To show how *FRProtector* achieves its objects in defending against control-flow hijacking attacks, we present our design of *FRProtector*, an effective way to detect the anomaly of control flow. In this section, we will begin with the design overview, and then present its detailed design.

3.1 Design Overview

We design *FRProtector* to support multiple security mechanisms to defend against control-flow hijacking attacks. The first one is function-level randomization. Under this mechanism, *FRProtector* reorders the locations of functions to increase the entropy of the program code segment in memory. *FRProtector* also supports control flow transfer protection. Randomization is a mitigating method which can be bypassed by some well-structured information disclosure, so we provide control flow transfer protection for deterministic defense. Under such mechanism, *FRProtector* marks each function with a unique label generated by XORing the stack frame pointer with the return address. Before the control flow transfer instruction (*ret* and indirect *jmp*) is executed, the program calculates the unique label with the same method, and then checking whether the two label is same. The overhead is lower if we compare the return address and the stack frame respectively. But taking into account the information disclosure, if the attacker gets the value of stored return address or stack frame in the register, then he can carefully build comparison labels to bypass the defense. But with the XOR method, it is very challenging that an attacker needs to change the value of the stack frame and the return address at the same time to meet the label which is calculated in the beginning of the function.

3.2 Function-Level Randomization

Today, most of the operating systems use ASLR to increase the difficulty of attackers to guess the layout of memory space. ASLR changes the base address

of the memory segment, making it difficult for an attacker to write the exploits directly with the results of static analysis. However, with the information disclosure, the attacker can bypass the ASLR through a memory address leakage. Function-level randomization achieves a more fine-grained code space layout randomization granularity, making the attacker to get the entire memory layout from memory leakage difficult. Thus, it mitigates the possibility of constructing a CRA with the slightest disclosure of information and static analysis.

Function-level randomization is mainly to reorder the location of functions. In an executable file, the code is stored in the code segment. When executing the file, the entire contents of a segment are stored in the (virtual) memory space. With ASLR, the offset of the code segment is different for every execution, but the relative locations of functions in memory do not change at all. With the function-level randomization, the relative orders of the functions will be disrupted, and function-level randomization may add some irrelevant instructions between functions such as NOP, which makes the entropy of the program to be further increased. Figure 2 shows an example of function layout in memory after function-level randomization. Therefore, with the help of function-level randomization, the gadget location obtained in the static analysis is no longer applicable. Attackers need to use other means (such as a lot of information disclosure) to get the gadgets.

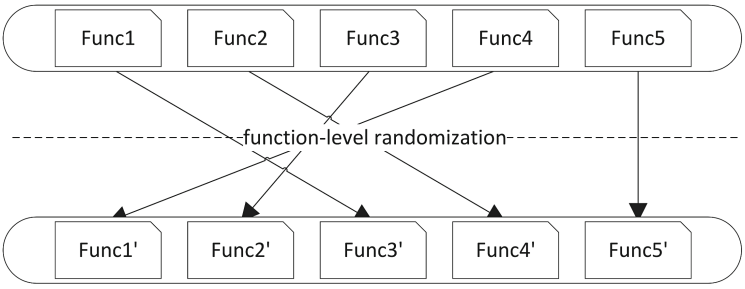


Fig. 2. Function-level randomization

There is a trade-off between security and performance when choosing at what time to do function randomization – reordering functions at loading time gives better security in that every execution of the program results in a different process memory image, but also adds more runtime overhead and bigger memory usage. *FRProtector* chooses to shuffle functions at compile time as we do not only depend on function randomization to defend against control-flow hijacking attack.

3.3 Protection of Control Flow Transfer

Now researchers mainly use CFI to prevent control flow hijacking. But CFI has been cautious about the problem of identification inaccuracy, compatibility and

overhead. *FRProtector* does not need to construct the CFG of a given program and it does not involve the correlation between functions, so that there is no compatibility problem between protected function and unprotected function.

In order to protect the control flow, *FRProtector* uses two mechanisms. First, in many cases, the attacker will hijack the control flow by modifying the return address, so the first mechanism is to detect whether the return address is changed. On the other hand, since the attacker may use other methods to hijack the control flow such as modifying the destination of indirect jump, *FRProtector* detects whether a function's internal execution flow is from the starting point of the function to the control flow transfer instruction.

Internal Control Flow Validation. As we know, a function consists of many basic blocks. There is a control flow transfer instruction at the end of each basic block. A function is executed from the entry block, and then the control flow is transferred to other basic blocks or functions. The mechanism for the internal control flow validation is to detect whether the control flow of the function is performed from the entry block. Therefore, we insert a random value that uniquely identifies the function at the entry block of the function, and check whether the re-calculated random value at *ret* and indirect *jmp* equals to the random value we inserted.

The random value is generated by XORing the stack frame pointer with the value of the return address rather than with a random value that is implemented in Stack Canaries [5]. This is because the control flow between functions is affected by the return address, and the return address can be used as a factor to see if the control flow is hijacked by modifying the return address.

Figure 3 shows the example that how function's internal control flow is protected. First, when Function 2 is called, *FRProtector* gets the value of the stack frame pointer and moves it into the register, and then XORs it with the value of the return address at the entry block of Function 2. Then it fetches and stores the value of the return address in another register and XORs it with the value of the stack frame before *ret*. Finally, *FRProtector* verifies whether the two register value are consistent. If true, the control flow will execute the *ret*, else the check_fail function will be executed. The detection point of this mechanism is before *ret* and indirect *jmp*, so it can detect both ROP and JOP.

4 Implementation

We have implemented *FRProtector* on top of the LLVM 3.9.1 compiler infrastructure [16]. *FRProtector* works on unmodified programs and supports Linux in 32-bit modes.

4.1 Function-Level Randomization

We implement the function-level randomization for *FRProtector* as an LLVM pass. The LLVM pass operates on the LLVM Intermediate Representation (IR),

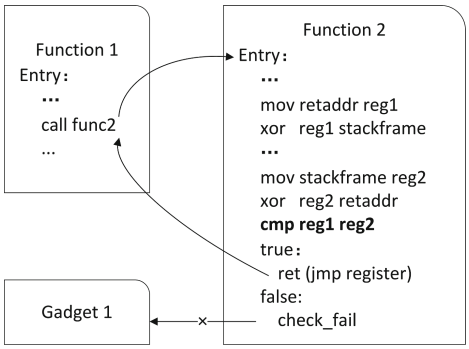


Fig. 3. Function's internal control flow detection

which is a low-level strongly-typed language-independent program representation. Although if we disrupt the location of the basic block the IR layer, the binary program will not change with the back-end compiler optimization. But the disruption of the location of the function will not be affected.

The function *CloneFunction* provided by LLVM can copy the information of a function into another function. Therefore, we randomize the order of the functions by copying them into other functions, deleting the original ones and then re-creating the new ones. We use a replacement algorithm to reorder the functions. For every function, a random number is created to decide the function that exchanges to.

This kind of randomization mechanism can be used to cloud environment. Multi-version of the randomization can make the applications have different memory layouts between offline version and the server side version. Therefore, it is harder for attackers to guess the addresses of gadgets.

4.2 Control Flow Transfer Protection

For control flow transfer protection, we use functions *llvm.returnaddress* and $llvm.frameaddress$ to get the return address and the value of the stack frame pointer respectively. When using llvm.address instructions twice in ONE basic block, llvm's back-end always reuses the first address value that the llvm.address has generated instead of getting the value from the stack twice. So in the implementation of this mechanism, we make a constant true transfer after retrieving the value generated by XORing the return address with the stack frame.

For example, the verify_password function has an unrestricted *strcpy* function, which can cause a buffer overflow to hijack control flow of the program. This function with *FRProtector* shown in Fig. 4 adds a check operation to see whether the XOR value is consistent with the value calculated at the beginning of the function before the return instruction. As the buffer overflow can modify the return address, so with our check at the end of the basic block the process will find errors and jump to *exit()* to quit execution for avoiding further losses.

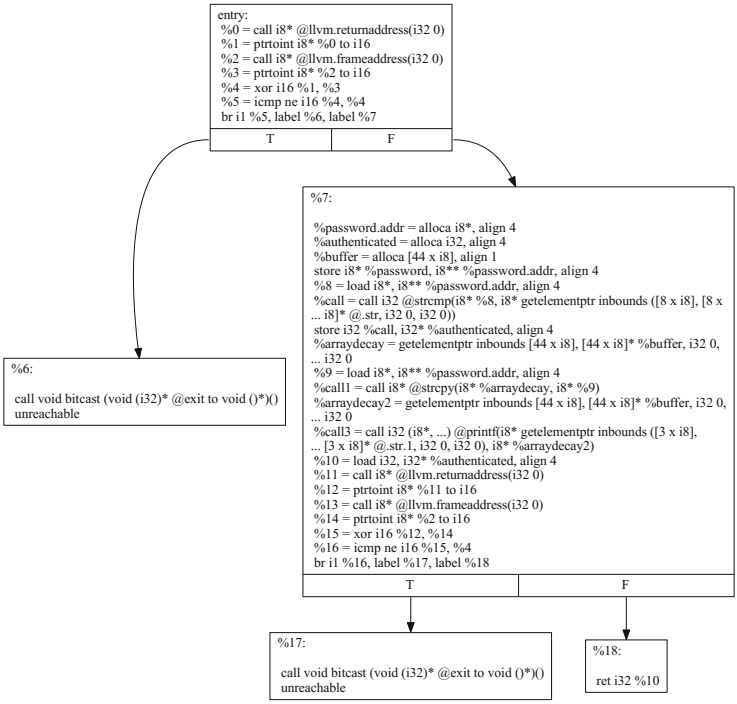


Fig. 4. verify_password function with FRProtector

5 Evaluation

In this section, we perform a number of experiments to demonstrate the effectiveness and efficiency of *FRProtector*. We experimentally show that *FRProtector* can effectively prevent all attacks that overwrite the return address in the RIPE benchmark. We evaluate the efficiency of *FRProtector* on SPEC CPU2006, and find average runtime overhead and space overhead are about 2% and 2.2% respectively.

All experiments were performed on a desktop computer with i7-4770 CPU running the x86 version of Ubuntu 16.04.

5.1 Effectiveness on the RIPE Benchmark

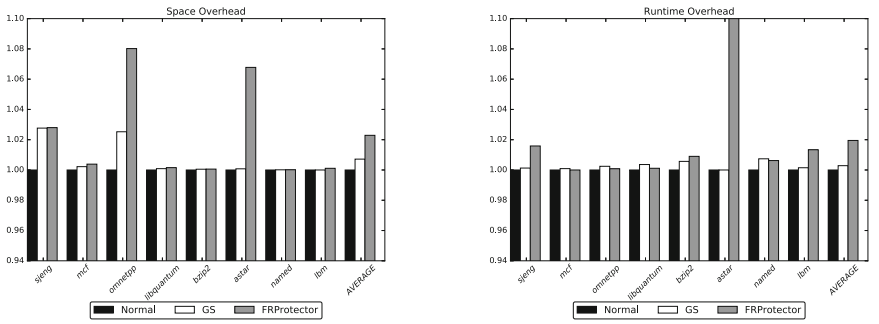
Runtime Intrusion Prevention Evaluator (RIPE) [14] is a benchmark test that detects all buffer overflow attacks. The goals of these attacks are to create files, Returntolibc and ROP. We find that there are 10 attacks which overwrites the ret instruction can be successfully launched with ASLR. When we use *FRProtector*, we find that all these 10 attacks are not available. So, we can effectively prevent the attack which overwrites the return address.

According to the design of *FRProtector*, the ret and indirect jump are both protected. So, if the attacker hopes to modify the destination address of the ret or indirect jump instruction in the middle of a function, *FRProtector* can detect it successfully.

5.2 Efficiency on SPEC CPU2006 Benchmarks

In this section, we evaluate the space overhead and the runtime overhead of StackGuard (*GS*) and *FRProtector*. We report numbers on SPEC CPU2006 benchmarks written in C and C++.

Space Overhead. Figure 5(a) shows the space overhead of our experiments with the benchmark programs. As shown, *GS* and *FRProtector* have nearly the same space overhead for most programs, it's for two reasons. First, both *GS* and *FRProtector* only insert several (about 4) instructions in a function, so the variation of the program is hairlike. Second, *FRProtector* protects more function than *GS*, but *GS* needs a function to create the random number. So, wane and wax, the space required is similar. The average space overhead of *FRProtector* is about 2.2%. For most programs, the space overhead experienced by *FRProtector* can be ignored. But some programs, such as *astar* and *omnetpp*, the space overhead is more than 6%. There are two reasons. First, we add instructions in every function, so the number of functions is an important factor. On the other hand, the checking instruction is inserted before every ret and indirect jump instruction, so the number of the transfer instructions also affects the space overhead.



(a) Space overhead of *FRProtector* (b) Runtime performance of *FRProtector*

Fig. 5.

Runtime Overhead. The runtime overhead of *FRProtector* is shown in Fig. 5(b). Results show that the average runtime overhead is about 2%. In [17], we know that the average performance overhead should be less than 5% when

the new method hopes to be used in industry. So, *FRProtector* could be adapted to industry requirements. *Astar* experiences much higher runtime overhead than other programs. We find this is due to the large number of short basic blocks it has makes the number of checking instructions increase. Meanwhile, the two registers *FRProtector* used to save the XOR value may reduce the number of registers available in a function. However, in the 64-bit program, the number of registers is increased, so the runtime overhead may be reduced.

Compared with *GS*, the runtime and space overhead of *FRProtector* only increased by about 1%. But with the disassembly file of the program, we find that *GS* protects less functions than *FRProtector*. It's due to *GS* is designed to protect functions that may have a stack vulnerability, while *FRProtector* hopes to check every function. *FRProtector* is less expensive than *GS* when adding protection to the same number of functions as *FRProtector* doesn't need to generate a random number and adds the similar number of instructions.

6 Discussion

In this section, we first compare *FRProtector* with CFI and Stack Guard, and then discuss the compatibility and limitations of *FRProtector*.

6.1 Comparison with CFI

FRProtector can be seen as one CFI method by enforcing policies for indirect *jmp* and *ret* instructions. For indirect *jmp*, *FRProtector* ensures the target of indirect jumps can be the entry of any functions by validating the control flow of a function must start from the entry block. The coarse-grained CFI has the same policy too. For *ret*, *FRProtector* ensures it must return to the corresponding caller by checking whether the target of a *ret* is overwritten. Therefore, *FRProtector* can achieve similar security compared with existing CFI approaches. Moreover, *FRProtector* does not need to analyze the source code or binary of a given program statically to compute the CFG.

6.2 Comparison with Stack Guard

Both *FRProtector* and stack guard (GS) introduce a random number. However, the role and the generation of the random number are different. Stack guard get the value by XORing a random number with the stack pointer, then the value is put between the return address and local variables in stack to detect whether the local variables' overflow overwrites the return address. The random number in *FRProtector* is the value generated by XORing the return address and the stack frame, then storing it in the register only. It can detect any attacks which change the return address, not just overflow. In addition, *FRProtector* also checks whether a function is executed from the entry basic block. GS just protects functions that may have buffer overflow vulnerabilities, while the protection is provided to all functions by default by *FRProtector*. *FRProtector* stores the

label in the register to increase the difficulty for the attacker to find it instead of storing it onto the stack. Finally, *FRProtector* also checks the label before indirect jump instructions to defend control-flow hijacking attacks that modify the registers used in indirect jump instructions.

6.3 Compatibility and Limitations

FRProtector is written on the IR layer of LLVM, and has nothing to do with source code. So it is source-level compatibility. It means that the problems caused by binary level protection such as function pointer errors do not occur in *FRProtector*. Since the function is the base unit for *FRProtector* in which *FRProtector* only checks whether the control flow is transferred from the first block, so it can be compatible with legacy libraries, functions and programs that do not enforce *FRProtector*.

The main limitation of *FRProtector* is it cannot defend against the control-flow hijacking that does not use the return or indirect jump instructions. For example, Counterfeit Object-Oriented Programming (COOP) [18] and Call Oriented Programming (COP) [19,20] use virtual functions and function calls respectively to achieve control flow hijacking. We leave it our future work – a more complete mechanism to defend against control flow hijacking.

7 Related Work

7.1 Function-Level Randomization

We implement function-level randomization in LLVM to mitigate the information disclosure and change the function's address for randomizing the return address. There are several techniques which have implemented function-level randomization. Marlin [21] is a bash shell that can randomize the target executable before launching it. It shuffles the functions in the executable code. Bin_FR [22] randomizes the binary directly, which adds random padding between functions and randomizes the order of functions. The advantage of Bin_FR is that it does not rely on the source code.

7.2 Compiler Techniques Counter Control Flow Hijacking

Lots of compiler techniques have been published to defend control-flow hijacking, especially to defend ROP. StackGuard [5] is an oldest method to prevent buffer overflow attacks by inserting a canary (random number) between the return address and local variables. G-free [23] is a compiler-based approach against ROP which uses the return address or indirect call/jump. Return-less [24] is a technique that aims to defend return-oriented rootkits (RORs). It replaces the return address in a stack frame into a return index and disallows a ROP to use it. It also proposes register allocation and peephole optimization to prevent legitimate instructions that happen to contain return opcode from being misused.

The stack pivot is an essential component in most ROP by modifying a stack pointer to point to the payload. PBlocker [25] is a technique which asserts the sanity of stack pointer whenever the stack pointer is modified to denial of stack pivot. But the stack pivot check can be bypassed by an attack mentioned in [26].

8 Conclusion

In this paper, we present *FRProtector*, a novel defense against control-flow hijacking. *FRProtector* implements the function-level randomization to increase the difficulty of the attacker to guess the code layout and also to change the return address of functions. On the other hand, *FRProtector* implements the control flow transfer protection by checking whether the address of transfer instruction has been modified, which can effectively protect the control flow of the program. *FRProtector* has implemented in LLVM. We evaluate *FRProtector* on SPEC CPU2006 and show that the average runtime overhead is 2% and the space overhead is 2.2% on average.

Acknowledgment. Supported by the National Natural Science Foundation of China (61373168, U1636107), and Doctoral Fund of Ministry of Education of China (20120141110002).

References

1. Heelan, S.: Automatic Generation of Control Flow Hijacking Exploits for Software Vulnerabilities (2009)
2. Andersen, S., Abella, V.: Data execution prevention. changes to functionality in microsoft windows XP service pack 2, part 3: memory protection technologies (2004)
3. Shacham, H.: The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In: ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, pp. 552–561, October 2007
4. Bletsch, T., Jiang, X., Freeh, V.W., Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In: ACM Symposium on Information, Computer and Communications Security, pp. 30–40 (2011)
5. Cowan, C., Pu, C., Maier, D., Hintony, H., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q.: StackGuard: automatic adaptive detection and prevention of buffer-overflow attacks. In: Conference on Usenix Security Symposium, p. 5 (1998)
6. PaX Team: Pax address space layout randomization (ASLR) (2003)
7. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity. In: ACM Conference on Computer and Communications Security, pp. 340–353 (2005)
8. Abadi, M., Budiu, M., Erlingsson, Ú., Ligatti, J.: Control-flow integrity principles, implementations, and applications. ACM Trans. Inf. Syst. Secur. (TISSEC) **13**(1), 4 (2009)
9. Zhang, M., Sekar, R.: Control flow integrity for cots binaries. In: Usenix Security, vol. 13 (2013)

10. Zhang, C., Wei, T., Chen, Z., Duan, L., Szekeres, L., McCamant, S., Song, D., Zou, W.: Practical control flow integrity and randomization for binary executables. In: 2013 IEEE Symposium on Security and Privacy (SP), pp. 559–573. IEEE (2013)
11. Mohan, V., Larsen, P., Brunthaler, S., Hamlen, K.W., Franz, M.: Opaque control-flow integrity. In: NDSS Symposium (2015)
12. Bittau, A., Belay, A., Mashtizadeh, A., Mazieres, D.: Hacking blind. In: IEEE Symposium on Security and Privacy, pp. 227–242 (2014)
13. Strackx, R., Younan, Y., Philippaerts, P., Piessens, F., Lachmund, S., Walter, T.: Breaking the memory secrecy assumption. In: European Workshop on System Security, Eurosec 2009, Nuremburg, Germany, pp. 1–8, March 2009
14. Wilander, J., Nikiforakis, N., Younan, Y., Kamkar, M., Joosen, W.: RIPE: runtime intrusion prevention evaluator. In: Twenty-Seventh Computer Security Applications Conference, ACSAC 2011, Orlando, Fl, USA, 5–9 December, pp. 41–50 (2011)
15. Snow, K.Z., Monrose, F., Davi, L., Dmitrienko, A.: Just-in-time code reuse: on the effectiveness of fine-grained address space layout randomization. In: Security and Privacy, pp. 574–588 (2013)
16. The LLVM compiler infrastructure. http://llvm.org/
17. Szekeres, L., Payer, M., Wei, T., Song, D.: SoK: eternal war in memory. In: IEEE Symposium on Security and Privacy, pp. 48–62 (2013)
18. Damm, C.H., Hansen, K.M., Thomsen, M.: Tool support for cooperative object-oriented design: gesture based modelling on an electronic whiteboard. In: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, pp. 518–525. ACM (2000)
19. Göktas, E., Athanasopoulos, E., Bos, H., Portokalidis, G.: Out of control: overcoming control-flow integrity. In: IEEE Symposium on Security and Privacy, pp. 575–589 (2014)
20. Sadeghi, A., Niksefat, S., Rostamipour, M.: Pure-call oriented programming (PCOP) chaining the gadgets using call instructions. J. Comput. Virol. Hacking Technol. **14**, 1–18 (2017)
21. Gupta, A., Habibi, J., Kirkpatrick, M.S., Bertino, E.: Marlin: mitigating code reuse attacks using code randomization. IEEE Trans. Dependable Secur. Comput. **12**(3), 1 (2015)
22. Fu, J., Zhang, X., Lin, Y.: Code reuse attack mitigation based on function randomization without symbol table. In: Trustcom, pp. 394–401 (2016)
23. Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., Kirda, E.: G-free: defeating return-oriented programming through gadget-less binaries. In: Computer Security Applications Conference, pp. 49–58 (2010)
24. Li, J., Wang, Z., Jiang, X., Grace, M., Bahram, S.: Defeating return-oriented rootkits with "return-less" kernels, pp. 195–208 (2010)
25. Prakash, A., Yin, H.: Defeating ROP through denial of stack pivot. In: Computer Security Applications Conference, pp. 111–120 (2015)
26. Yan, F., Huang, F., Zhao, L., Peng, H., Wang, Q.: Baseline is fragile: on the effectiveness of stack pivot defense. In: IEEE International Conference on Parallel and Distributed Systems, pp. 406–413 (2016)