International Journal of Applied Engineering Research ISSN 0973-4562 Volume 12, Number 17 (2017) pp. 6988-6996

Improved Technique for Verification of Call-return Flow Integrity for Compiler-based Defense against Return-oriented Programming Attacks

Younsung Choi

Department of Cyber Security, Howon University, 64 Howon University 3 Gil, Impi-Myeon, Gunsan-Si, Jeonrabuk-Do, 54058, Republic of Korea.

Orcid ID: 0000-0002-3185-8670, Scopus Author ID: 56173148800

Abstract

Return-oriented programming (ROP) attacks have been increasing in number recently. ROP is an exploitation technique that can bypass non-executable page protection methods by using existing codes within benign programs or modules. There has been much research on defense methods against ROP attacks, but most of them have high performance overhead (dynamic instrumentation approach) or high time complexity (compiler-based approach) in terms of the detection of gadgets. The ROP defense technique recently proposed by Lee *et al.* has overcome the limitations of the compiler-based approach, and has further proved its efficiency. Their defense technique performs calculations with a single global variable immediately before the execution of each ret instruction (-1) and at the resetting position (+1). Moreover, their defense technique achieved $O(1)$ in detection time complexity by detecting gadgets within only two executions. In their experiment, the performance overhead was 1.62% and the file size overhead was 4.60%. To verify the control-flow integrity (CFI), their defense scheme was performed to simplify codes by computing a special variable, check_value. However, the code is inefficient because it performs the same computation on all call and ret instructions. In this paper, we propose an ROP defense method to verify the CFI through concentrated functions. We improved Lee *et al.*'s method by allowing a specific function to separately execute the calculation of the check_value. Our experiment could reduce the file size overhead by approximately 40% while maintaining the efficiency, detection time complexity, and performance presented in Lee *et al.*'s defense technique.

INTRODUCTION

A return-oriented programming (ROP) attack, also known as a code reuse attack, is a combination of codes that reside on a normal program or module. The ROP attack disables the latest operating system security functions such as Data Execution Prevention (DEP) or $X \oplus W$ (write XOR execute) for non-executable page protection. The basic unit of an ROP attack is a gadget, a small piece of code that ends in a ret instruction, mainly using a benign program or module instruction that is running on the operating system. Furthermore, after chaining these gadgets together for consecutive execution, the start pointer set of the gadgets is injected into a program where the vulnerability such as stack overflow exists. Subsequent ROP attacks execute gadgets consecutively, starting with the initial return address modulation. This destroys the execution flow of normal code and illegally allocates an executable memory area, causing malicious behavior. Particularly, when accessing a code area of a program image loaded in memory, this attack can access not only an instruction intended by a programmer or a compiler but also an unintended instruction. Thus, the ROP attack is *Turing complete*.

Numerous studies have been proposed to defend ROP attacks, which are largely divided into four approaches. The first approach is *dynamic instrumentation*. This approach tracks each instruction of a program at run-time to identify the gadgets. Most of the studies proposing this approach require additional dynamic binary instrumentation (DBI) tools to track instructions, which incur a high performance overhead (maximum 219%).

The second approach is *randomization*. This approach maps a particular section (.text) of an executable file onto a random address at each execution to randomize the address of the virtual memory, such as address space layout randomization (ASLR). The approach provides high entropy. It can also indirectly defend against ROP attacks because they are based on knowledge of the location and structure of codes, owing to the nature of ROP attacks that reuse codes. However, their applicability is limited due to a high performance overhead, and the fact that non-randomized dynamic libraries are vulnerable to ROP attacks.

The third is a *network-based approach*. This approach detects ROP attacks by loading incoming packets from the network onto a static stack frame, and mapping the packets onto the address of the module loaded in the virtual memory. However, this approach can cause network latency problems due to its monitoring of many packets that may not be ROP attacks. It also incurs a high performance overhead due to address mapping with modules loaded into the virtual memory.

Finally, the fourth is a *compiler-based approach* that applies the defense code against ROP attacks at program compile-time. This approach is similar to a program's control-flow integrity

(CFI) approach. The main advantage of this approach is that it can detect the gadgets and defend the attacks in a stand-alone fashion without the help of a third-party tool. Therefore, this approach has a very low performance overhead compared to other approaches. However, the file size of the program increases (by approximately 30%) because the defense code is applied to the program itself. Furthermore, the approach is disadvantageous in that it cannot identify ROP gadgets using unintended instruction sequences, because the code for defense is determined at compile time.

The recently proposed defense mechanism in the compiler-based approach is a *zero-sum defender*. Their defense technique computes +1 at the prologue of the function, and -1 at the epilogue, to verify a pair of call and ret instructions with a single global variable, call_level. The defense is performed by checking whether the call_level variable is negative when an abnormal return is performed by an ROP attack. Their defense technique proved its efficiency by achieving a very low overhead for performance and file size growth (performance overhead 1.7%, file size overhead 4.5%). However, their method cannot provide defense if the value of call_level is greater than the number of gadgets. In the detection time complexity of the ROP gadget, the zero-sum defender is $\mathcal{O}(n)$, where n is the number of gadgets). Particularly, their possibility of failure to defend the ROP attacks can be even higher because the compiler-based approach cannot identify the gadgets that use unintended instruction sequences. It is a very difficult task for an attacker to combine gadgets for ROP attacks. For this reason, for a successful ROP attack, an attacker combines all the intended and unintended instruction sequences. Therefore, even if the compiler-based approach can only identify the gadget composed of the intended instruction sequences, the approach can more effectively (compared to other approaches) defend against the ROP attack itself, by reducing the detection time complexity.

The defense scheme recently proposed by Lee *et al.* has overcome the limitations of the compiler-based approach, and has further proved its efficiency by improving the zero-sum defender. Their defense technique performs calculations with a single global variable immediately before the execution of ret instruction (-1) and at the resetting position (+1). Moreover, their defense technique achieved 0 (1) in detection time complexity by detecting gadgets within only two executions. This technique overcomes the limitations of the compiler-based approach that fails to identify gadgets composed of unintended instruction sequences. However, the defense code added to the compile-time in their defense method generates redundant code to compute the control variable check_value. The defense code is added to the original program in proportion to the number of call and ret instructions, thus increasing the file size overhead.

In this paper, we propose a defense method, using the function-concentrated to compute the CFI, against ROP attacks. Our method is based on Lee *et al.*'s scheme. We achieved a lower file size overhead by concentrating the iterations of the same computation in one location to verify the integrity of the pair of call and ret instructions. Our method reduced the file size overhead by 40% or more.

The remainder of this paper is organized as follows. In Section 2, we explain relevant existing knowledge to elucidate our proposed method. Lee *et al.*'s ROP defense scheme is reviewed in Section 3. We provide a detailed description of our proposed method in Section 4. In Section 5, we discuss the experiments conducted on the proposed system. Our discussion and future work is provided in Section 6, and Section 7 presents the conclusion of our study.

PRELIMINARIES

Procedure Linkage Mechanism

The procedure linkage is a contrast between the compiler, the operating system, and the target machine, that clearly divides responsibility for naming, allocation of resources, addressability, and protection. The linkage convention is machine dependent. For example, it depends implicitly on information such as the number of registers available on the target machine and the mechanisms for executing a call and a return.

Figure 1 shows how the pieces of a standard procedure linkage fit together. Each procedure has a prologue and an epilogue. Each call site includes both a *pre-call* and a *post-return*. When a program is executed, each procedure creates a stack frame to control local variables. The prologue of the procedure plays the role of creating a stack frame, and the epilogue plays the role of organizing the stack frame. EBP and ESP are the registers that enable the procedure. Furthermore, pre-calls are associated with the way arguments are passed when a procedure is called, and post-calls are associated with return values when a procedure completes the execution. This may vary depending on the compiler or calling convention.

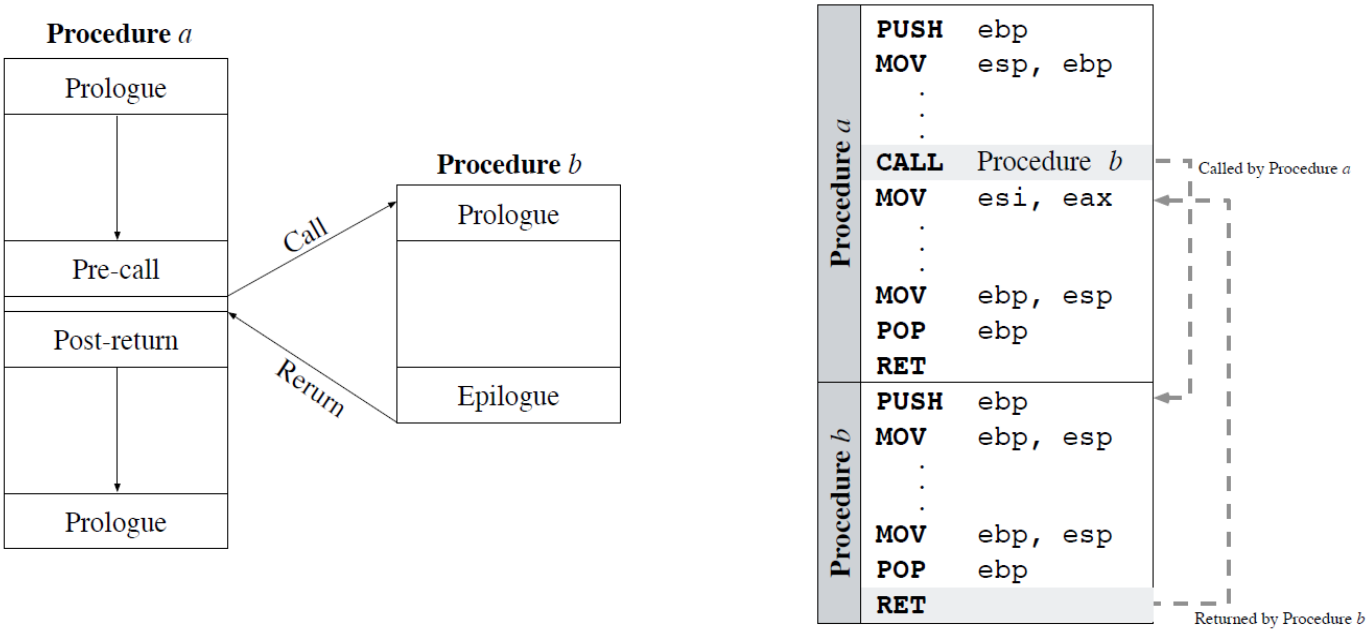


Figure 1: A standard procedure linkage

We focus on the relationship between call and ret instructions in the procedure linkage. The call instruction transfers the flow of the program to the procedure. In addition, the call instruction differs from the jmp instruction because a call saves a return address on the stack. The instruction pointer (IP or EIP register) always points to the next instruction in the program. For the call instruction, the contents of IP or EIP (x86) are pushed onto the stack; therefore, program control passes to the instruction following the call after a procedure ends. For 80386 and above, the ret instruction retransfers a 32-bit number from the stack and places it into EIP. When EIP is changed, the address of the next instruction is at a new memory location. This new location is the address of the instruction that immediately follows the most recent call to a procedure.

EIP is indirectly controlled through commands such as call, ret, and jmp. This is because the microprocessor typically does not allow direct access to the EIP register through the command interface, due to various security problems.. Particularly, the ret instructions is abused by an attack such as ROP because such attacks use the value on the stack to set EIP, which is called *return without call*. If the return address on the stack can be arbitrarily manipulated by an attacker, the EIP could be manipulated by the attacker regardless of the intention of the program. Next, we describe the ROP attacks.

Return-oriented Programming Attack

The first published exploit that reuses existing code for a *return-into-libc* attack has been presented by Solar Designer in 1997 [18]. The exploit overwrites the original return address to point to a critical library function. Specifically, it targets the system() function of the standard UNIX C library libc, which is linked to nearly every process running on a UNIX-based system.

The return-into-libc attack technique has some limitations compared to classic code injection attacks. First, an adversary is dependent on critical libc functions such as system(), exec(), or open(). Hence, if we either instrument or eliminate these functions, an adversary would no longer be able to perform a reasonable attack. In fact, one of the first proposed defenses against return-into-libc is based on the idea of mapping shared libraries to memory addresses that always contain a NULL byte [18]. Second, return-into-libc only allows calling one function after each other. Hence, an adversary is not able to perform arbitrary malicious computation. In particular, it is not possible to perform unconditional branching.

Shacham generalizes the idea of borrowed code chunks exploitation by introducing return-oriented programming [1]. This attack technique tackles the previously mentioned limitations of return-into-libc attacks. The basic idea is to execute a chain of short code sequences rather than entire functions. Multiple code sequences are combined to a so-called gadget that performs a specific atomic task, e.g., a load, add, or branch operation. Given a sufficiently large code base, an adversary will most likely identify a gadget set that forms a new Turing-complete language. That said, the derived gadget set can be exploited to induce arbitrary malicious program behavior. The applicability of ROP has been shown on many platforms including x86 [1], SPARC [19], Atmel AVR [20], PowerPC [21], ARM [22], and z80 [23].

The basic idea and workflow of a ROP attack is shown in Figure 2.

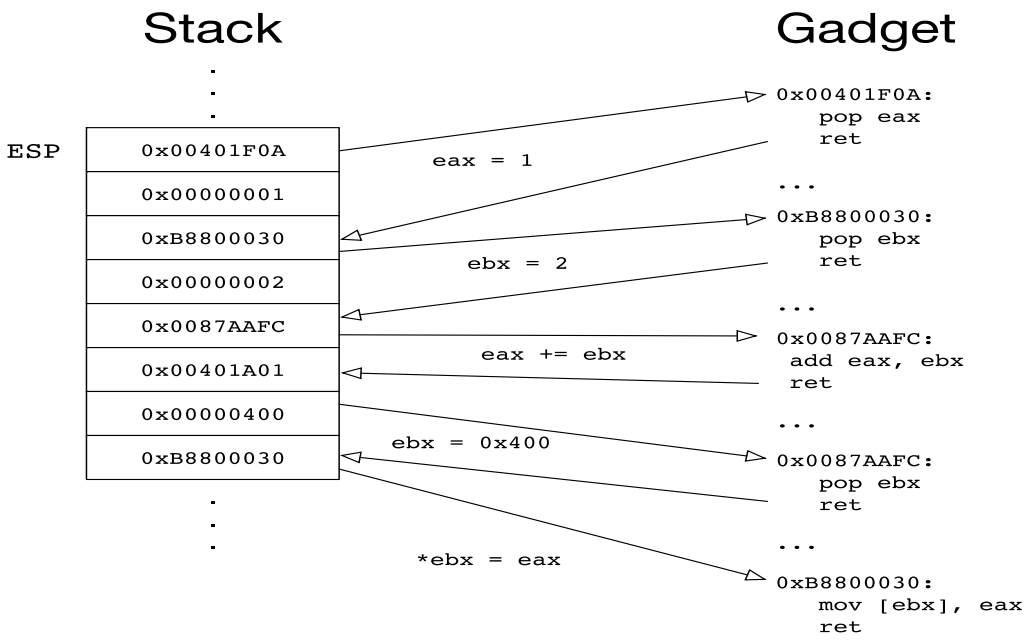


Figure 2: Basic principle of return-oriented programming attacks

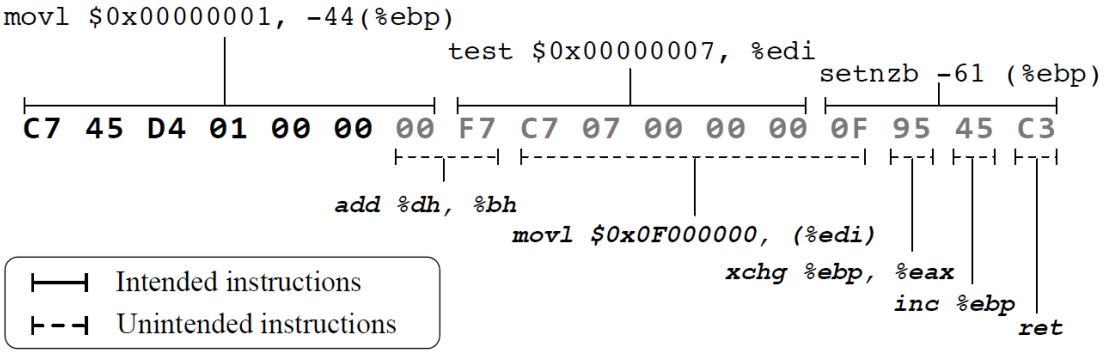


Figure 3: Example for a gadget using an unintended instructions sequence [24]

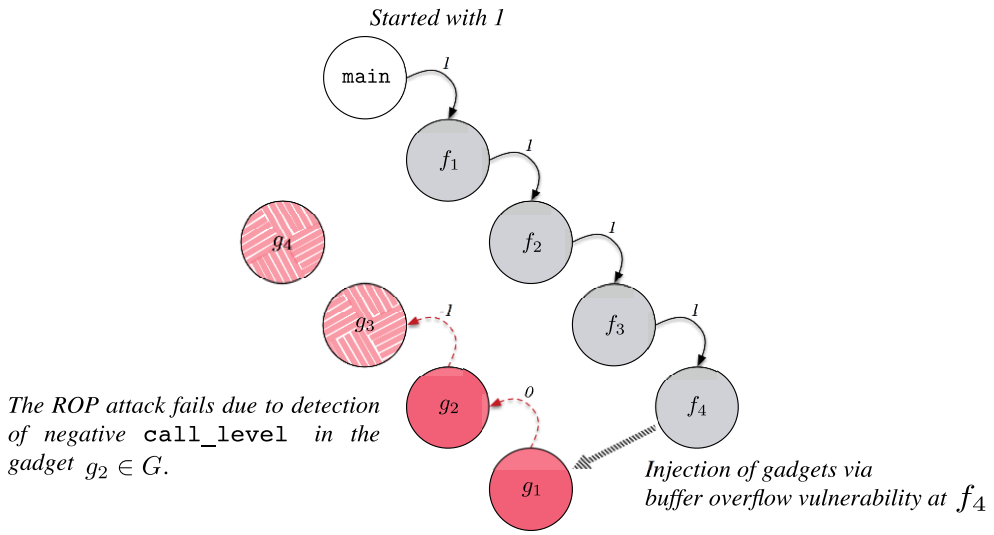


Figure 4: Lee *et al.*'s proposed scheme to defend against ROP attacks

A crucial feature of ROP on x86 is the invocation of the so-called unintended instruction sequences. These can be issued by jumping into the middle of a valid instruction resulting in a new instruction sequence neither intended by the programmer nor the compiler. Such sequences can be found in large number on the x86 architecture due to unaligned memory access and variable-length instructions. The detailed example of an unintended instructions sequence is shown in Figure 3. In the intended instruction sequence, the C3 byte is part of the last instruction. However, if the interpretation starts six bytes later, the C3 byte will be interpreted as a return instruction.

LEE *et al.*'S DEFENSE SCHEME REVIEW

In [16], a regular return is verified by using a special variable named check_value. The code for this verification is included at compiler-time. When the program begins, the check value has an initial value of 1. During the operation of the program, the value of check_value is reduced by 1 before the execution of the return and is increased by 1 immediately after returning to the caller. In other words, a value of 1 immediately proceeds a ret instruction and a value of +1 immediately follows a call instruction.

In Figure 4, we show how their improved method rapidly detects and defends gadgets despite being under the same conditions in [13].

While moving in a continuous call-flow until f_4, where the weakness is present, their check_value is maintained at the value of 1 which is initially set. Moreover, rapid detection is possible since the maintained check_value of 1 in a normal call-flow can be a negative value (-1) through just two abnormal return-flows. To safely support *multi-threaded applications*, the check_value is generated as a *static thread-local variable*. Their scheme, regardless of the number of gadgets, can identify two gadget execution, thus having a time complexity of O(1) in even the worst case. For their proposed defense method, the experimentally demonstrated performance overhead was 1.62% and the file size overhead was 4.6%. Compared to prior studies, their results obtained satisfactory performance overhead and file size overhead. However, they insert the duplicated code that calculates check_value with the original program to identify the gadgets. Generally, the static thread-local variable is reserved in a special areas (e.g., in the Windows case, .tls section) at compile-time. The thread-local variable cannot directly refer via any instructions. Hence, after it copies into to same thread area, the thread-local variable must be referred by register or stack memory. This problem causes an increase in the code that simply computes +1 and -1. Moreover, the increased code is redundant and leads to the file size overhead. In this paper, we improve their drawback, and detailed in next section.

OUR PROPOSED METHOD

In this section, we propose a new compiler-based defense method against ROP attacks. Our defense scheme is based on Lee's scheme. As mentioned before, Lee's defense scheme performs simple computation with a special variable check_value, but the defense codes are incremented and duplicated by the number of call and ret instructions. Our main contribution is to improve the file size overhead through concentrated functions, computing the CFI with a special variable check_value.

Figure 5 shows the workflow in our proposed method against ROP attacks. In our method, the initialization of the check_value is similar to Lee *et al.*'s defense scheme.

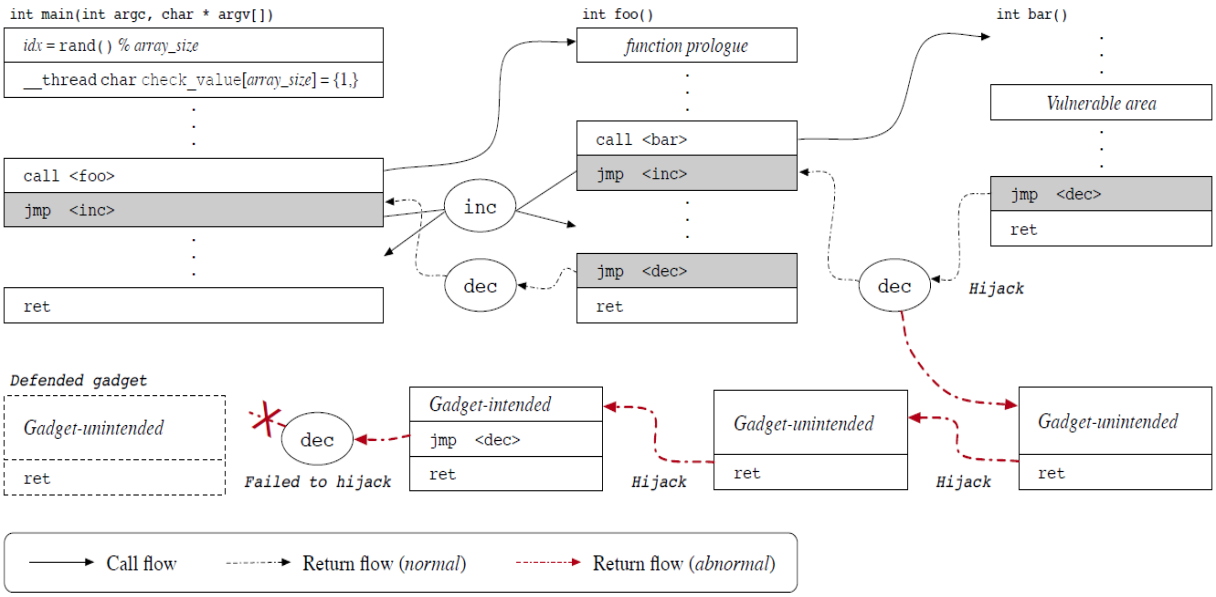


Figure 5: Our proposed workow to identify the gadgets in ROP attacks

When some function is executed in a multiple-thread environment, updating the control variable and check_value could interfere with concurrent execution, and there is a possibility of our defense method's failure in ROP attacks. To safely support multi-threaded applications, we represent check_value as a thread-local variable. In addition, our control variable check_value applies a randomization technique, similar to that used by Lee *et al.*'s defense scheme and the zero-sum defender. This is to prevent frequent modulation of malware accessing memory addresses with fixed constant values.

In Figure 5, we show how our improved method rapidly detects and defends against gadgets in an ROP attack. We insert commands that transfer control to our concentrated functions inc and dec immediately before the return instruction, and immediately below the call instruction (i.e. return address). The defense scheme by Lee *et al.* inserted instructions for calculating the check_value at this location. As previously mentioned, the check_value is stored as a static thread-local variable and must be transferred to register at run-time for further use, which eventually leads to an increase in defense codes. However, the calculation of check_value in our method is concentrated on inc and dec functions. The call and ret instructions only transfer control flow, to the inc and dec functions respectively, through jmp instructions. The jmp instruction occupies only 5 bytes in the 32-bit OS environment.

Similar to [1], our scheme is also identifiable in execution of two gadgets that consist of an intended instruction sequence; thus, having a time complexity of $\mathcal{O}(1)$ in even the worst case. Although return flow is normally operated through repetitively calculating 0 and 1, it is defendable by creating a negative value in the case of abnormally returning flows, such as with gadgets.

In Figure 6, we show the functions that were concentrated in computing the CFI with a special variable check_value. These functions, named dec and inc, are involved in verifying the matched pair of call and ret instructions. A dec function is called to calculate -1 for check_value immediately before the ret instruction is executed, and a dec function is further called for the +1 calculation at the position returning to the caller.

The check_value is generated as a static thread-local variable. Typically, because static thread-local variables cannot be referenced directly in the code section of the executable file image, they are referenced using the segment register (shown in Figure 6). Such use eventually leads to an increase in the original file size, which grows in proportion to the number of call and ret instructions in Lee's scheme. In our proposed method, the calculation of the check_value is concentrated on dec and inc functions. In each of the call and ret instructions, only commands that transfer control to dec and inc functions are inserted.

Generally, the jmp instruction performs an unconditional jump in the x86 assembly language. It transfers the flow of execution by changing the instruction pointer register. We insert the jmp instruction immediately before the ret instruction, and immediately after the call instruction (location returning from the callee). This calls the dec and inc functions, respectively, and performs a calculation on check_value. The jmp instruction takes up only 5 bytes of space in a 32-bit environment. This method is more efficient than Lee's method to insert codes to calculate check_value, a static thread-local variable for every call and ret instruction.

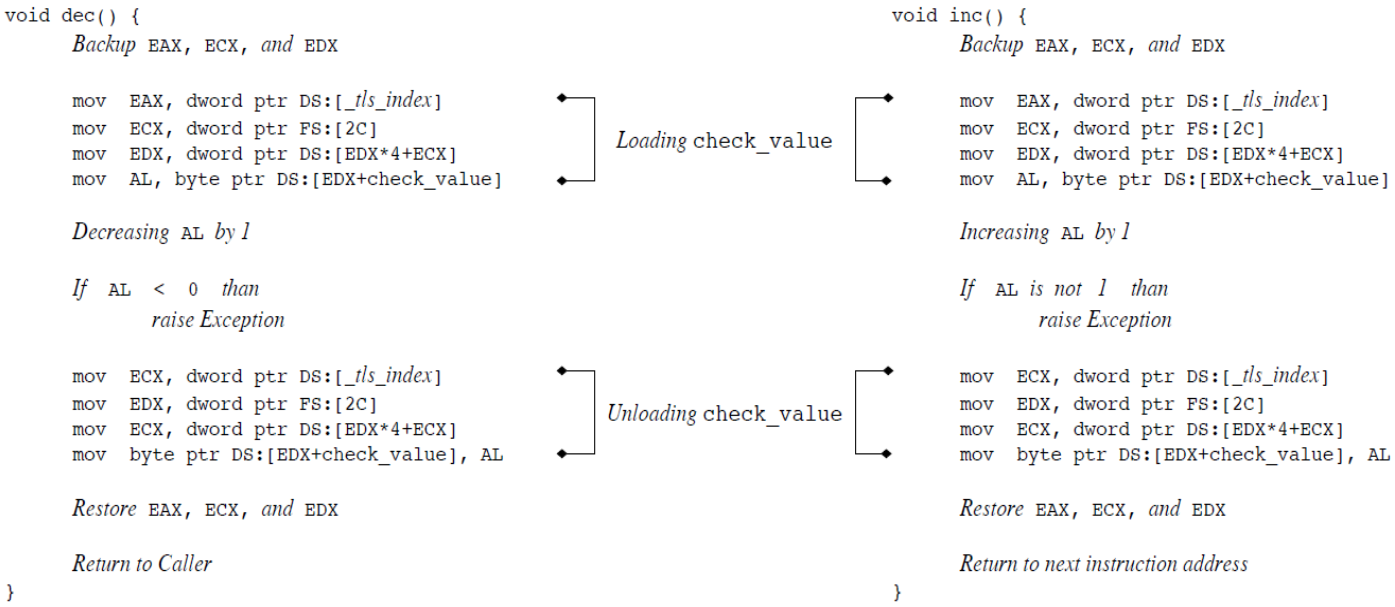
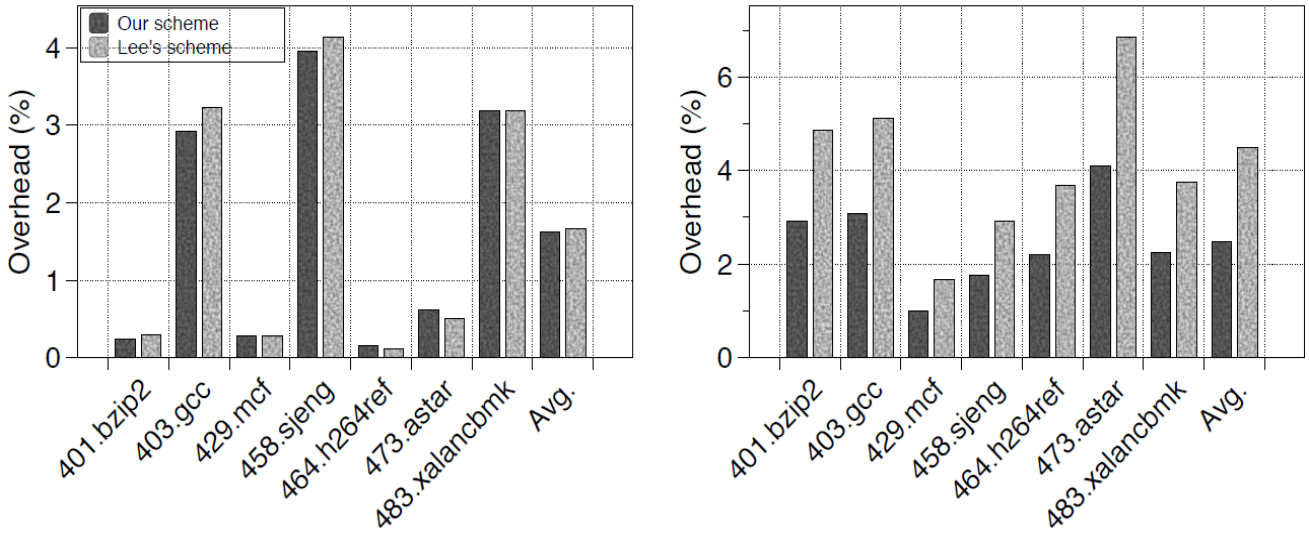


Figure 6: Our proposed dec() and inc() functions concentrated for defending against ROP attacks



(a) Performance overhead. (b) File size overhead.

Figure 7: Overhead of our scheme and Zero-sum defender

EXPERIMENTS

Our proposed method was implemented based on LLVM 3.2. We performed our experiment using the Linux kernel 3.2 on a 2.6 GHz Intel Core i7 machine with CPU-intensive performance benchmark suite SPEC CPU INT 2006. We measured the performance overhead and the file size overhead, and the results were compared to those of the existing Lee *et al.*'s scheme and zero-sum defender scheme.

In previous research, the dynamic instrumentation approach had an average performance overhead of 217%. Additionally, the average values of the file size overhead were 73% and 30% for the randomization approach and the compiler-based approach, respectively. The recently proposed zero-sum defender achieved a very rapid and effective defense with a 1.67% performance overhead and a 4.50% file size overhead. Experimental results in Lee *et al.*'s scheme were comparable to the zero-sum defender. The difference between the approaches is that the detection time complexity for the ROP gadget is of $\mathcal{O}(n)$ complexity using the zero-sum defender, and of $\mathcal{O}(1)$ complexity using the scheme by Lee *et al.*

As shown in Figure 7, the performance overhead and file size overhead in our experiment were 1.62% and 2.47%, respectively, which demonstrated that our scheme was as fast and effective as the existing zero-sum defender and Lee *et al.*'s scheme. Our ROP defense scheme is based on the scheme by Lee *et al.*, and thus, maintains their detection time complexity of $\mathcal{O}(1)$. However, our scheme reduced the file size overhead of the scheme by Lee *et al.* by 40% or more.

DISCUSSION

Intel introduced shadow stack technology to verify the return address when executing call and ret instructions on a microprocessor. A shadow stack is a second stack for the program that is used exclusively for control transfer operations. This stack is separate from the data stack, and can be enabled for operation individually in user mode or supervisor mode. When shadow stacks are enabled, the call instruction pushes the return address onto both the data and shadow stacks. The shadow stack technique was applied from the 7th generation of Intel microprocessors. This technique may be most effective in defending ROP attacks because it validates the CFI in microprocessor or hardware. However, there are cost issues associated with changing the microprocessor on all systems. For systems equipped with System-on-a-chip (SoC) type microprocessors, the cost problem is more serious because the machine itself must be replaced.

Our ROP defense method can be applied independently to each program at compile-time, and does not require a separate stack memory space. Furthermore, our CFI verification method, through the calculation of our single variable check_value, is more performance and memory efficient than the verification method of return addresses stored by consecutive function calls. For example, when 10 consecutive function calls occur and a total of 10 return addresses are stored in the shadow stack, the shadow stack takes 40 bytes of space in a 32-bit environment. However, our method requires only the space (4 bytes) of the check_value. Additionally, in terms of the verification, the shadow stack method must verify different return addresses each time a return instruction is executed, whereas our method needs only to check the negative value.

CONCLUSION

Return-oriented programming (ROP) is an exploitation technique that can bypass non-executable page protection methods by using existing codes within benign programs or modules. The defense scheme recently proposed by Lee *et al.* has overcome the limitations of the compiler-based approach, and has further proved its efficiency by improving the zero-sum defender. Their defense technique performed calculations with a single global variable immediately before the execution of ret instruction (-1) and at the resetting position (+1). Moreover, their defense technique achieved $\mathcal{O}(1)$ in detection time complexity by detecting gadgets within only two executions. In this paper, we proposed a method to reduce the file size overhead based on Lee's scheme. In our method, the calculation of the special variable check_value to verify the CFI was concentrated on the dec and inc functions. In our experiment, the performance overhead was 1.62%, and the file size overhead was 2.47%. This resulted in a reduction of file size overhead by approximately 40% while maintaining the advantages of Lee's defense method.

ACKNOWLEDGMENTS

This work was supported by the National Research Foundation of Korea grant funded by Korea government (Ministry of Science, ICT & Future Planning) (NRF-2017R1C1B5017492) and this research was supported by financial support of Howon University in 2017.

REFERENCES

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

[2] Roemer, R., Buchanan, E., Shacham, H., & Savage, S. (2012). Return-oriented programming: Systems, languages, and applications. ACM Transactions on Information and System Security (TISSEC), 15(1), 2.

[3] T. B. Matt Miller and M. Howard, “Mitigating software vulnerabilities," Microsoft Corporation, Tech. Rep., July 2011. [Online]. Available: http://www.microsoft.com/download/en/details.aspx ?displaylang=en&id=26788

[4] S. Designer, “Getting around non-executable stack (and fix),” 1997.

[5] J. McDonald, “Defeating solaris/sparc non-executable stack protection,” Bugtraq, Mar, 1999.

[6] Schwartz, E. J., Avgerinos, T., & Brumley, D. (2011, August). Q: Exploit Hardening Made Easy. In USENIX Security Symposium (pp. 25-41).

[7] Davi, L., Sadeghi, A. R., & Winandy, M. (2011, March). ROPdefender: A detection tool to defend against return-oriented programming attacks. In Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security (pp. 40-51). ACM.

[8] Bhatkar, S., DuVarney, D. C., & Sekar, R. (2005, August). Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In USENIX Security Symposium (pp. 271-286).

[9] Wartell, R., Mohan, V., Hamlen, K. W., & Lin, Z. (2012, October). Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code. In Proceedings of the 2012 ACM conference on Computer and communications security (pp. 157-168). ACM.

[10] Choi, Y., & Lee, D. (2015). Strop: Static approach for detection of return-oriented programming attack in network. IEICE Transactions on Communications, 98(1), 242-251.

[11] Li, J., Wang, Z., Jiang, X., Grace, M., & Bahram, S. (2010, April). Defeating return-oriented rootkits with return-less kernels. In Proceedings of the 5th European conference on Computer systems (pp. 195-208). ACM.

[12] Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., & Kirda, E. (2010, December). G-Free: defeating return-oriented programming through gadget-less binaries. In Proceedings of the 26th Annual Computer Security Applications Conference(pp. 49-58). ACM.

[13] Kim, J., Kim, I., Min, C., & Eom, Y. I. (2014). Zero-sum defender: Fast and space-efficient defense against return-oriented programming attacks. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, 97(1), 303-305.

[14] Kayaalp, M., Schmitt, T., Nomani, J., Ponomarev, D., & Abu-Ghazaleh, N. (2013, February). SCRAP: Architecture for signature-based protection from code reuse attacks. In High Performance Computer Architecture (HPCA2013), 2013 IEEE 19th International Symposium on (pp. 258-269). IEEE.

[15] Göktaş, E., Athanasopoulos, E., Polychronakis, M., Bos, H., & Portokalidis, G. (2014, August). Size does matter: Why using gadget-chain length to prevent code-reuse attacks is hard. In Proceedings of the 23rd USENIX conference on Security Symposium (pp. 417-432). USENIX Association.

[16] Lee, D., Jung, J., Choi, Y., & Won, D. (2016). Improvement and Weakness of Zero-Sum Defender against Return-Oriented Programming

Attacks. IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences, 99(12), 2585-2590.

[17] Cooper, K., & Torczon, L. (2011). Engineering a compiler. Elsevier.

[18] Designer, S. (1997). lpr LIBC RETURN exploit.

[19] Buchanan, E., Roemer, R., Shacham, H., & Savage, S. (2008, October). When good instructions go bad: Generalizing return-oriented programming to RISC. In Proceedings of the 15th ACM conference on Computer and communications security (pp. 27-38). ACM.

[20] Francillon, A., & Castelluccia, C. (2008, October). Code injection attacks on harvard-architecture devices. In Proceedings of the 15th ACM conference on Computer and communications security (pp. 15-26). ACM.

[21] Lindner, F. (2009). Router exploitation. Black Hat, http://www.blackhat. com/presentations/bh-usa-09/LINDNER/BHUSA09-Lindner-RouterExploit-SLIDES. pdf.

[22] Lian, W., Shacham, H., & Savage, S. (2015). Too LeJIT to Quit: Extending JIT Spraying to ARM. In NDSS.

[23] Checkoway, S., Feldman, A. J., Kantor, B., Halderman, J. A., Felten, E. W., & Shacham, H. (2009). Can DREs Provide Long-Lasting Security? The Case of Return-Oriented Programming and the AVC Advantage. EVT/WOTE, 2009.

[24] Buchanan, E., Roemer, R., Savage, S., & Shacham, H. (2008). Return-oriented programming: Exploitation without code injection. Black Hat, 8.

[25] Intel Corporation, “Control-flow Enforcement Technology Preview," pp. 1-136, Jun. 2016.