Monitoring Control Flow History To Detect Code Reuse Attacks

SOO-YOUNG KIM
KAIST
Department of Computer Science
291 Daehak-ro, Yuseong-gu, Daejeon
South Korea
sooyoungkim@kaist.ac.kr

GYUNGHO LEE
Korea University
Department of Computer Science and Engineering
145 Anam-ro, Seongbuk-gu, Seoul
South Korea
ghlee@korea.ac.kr

Abstract: Code Reuse Attacks (CRAs) are devastating security exploits that allow attackers to produce a malicious result by reusing the existing code. One such technique, return-oriented programming, is based on arbitrary execution of specific code snippets (called gadgets) from a stack corruption exploit. Although CRAs can be prevented by full control flow integrity (CFI) checking, software CFI checking has significant overhead and is vulnerable to unintended branch instructions. In this paper, we propose Miss Verification Unit (MVU), a light-weight security module that can address the limitations of CFI. In order to provide a protection mechanism with low overhead, a branch predictor and branch target buffer is used to narrow down the validation scope. MVU checks the validity only in case of branch miss, assuming that a abnormal control flow incurs CRAs. With this simple checking mechanism, MVU only incurs 2% overhead for programs from the SPEC CPU integer benchmark.

Key–Words: Computer Architecture, Security, Trustworthy Computing

1 Introduction

Computers are exposed to constant threats of software attacks. Researchers of computer security society have explored ways to provide complete trustworthiness of computers [7, 13], yet it is still an on-going work. Alternately, there have been numerous solutions that successfully block certain flows of the attack itself. Although they can block targeted attacks, these partial solutions lead attackers launch new attacks that circumvent the secure area.

Code injection attacks are an early attempt to execute malicious code by injecting code into a program. However, it is no longer threatening since the introduction of protection mechanism such as $W \bigoplus X$. In order to circumvent such techniques, code reuse attacks (CRAs) are proposed as a response. CRAs can execute malicious code by reusing the existing code. For example, the return-into-libc attack calls existing libc fuction, and return-oriented programming (ROP) forms gadgets (small snippets that end with return) using existing instructions. Although there have been several defense techniques[5, 8, 12], newly proposed technique, jump oriented programming (JOP) eliminate the necessity of return instruction. Although control-flow integrity (CFI) can prevent both ROP and JOP, they include high performance overhead that cannot be commercialized.

To address this problem, we propose Miss Verification Unit (MVU), a lightweight hardware-supported protection scheme against code reuse attacks (CRAs). It is an efficient security unit that verifies the instructions that are mispredicted by a branch predictor. By checking anomalies of regenerated addresses, MVU showed 95% accuracy in CRA detection, with 2% overhead on average.

The remainder of this paper is organized as follows. Section 2 describes the background and related work. MVU is introduced in Section 3, and its performance is evaluated in Section 4. Section 5 provides the discussion of MVU. Finally, Section 5 offers some concluding remarks.

2 Background

2.1 Buffer Overflow and Code Injection Attack

Buffer overflows are one of the most common software exploits in languages without type safety such as C/C++. Stack smashing is a buffer overflow attack, which allows the attacker to overwrite the return address of the function with the address of the malicious injected code. Several approaches were developed to defeat stack smashing attacks [2, 6].

Nevertheless, buffer overflow vulnerabilities remain prevalent. Hardware solutions have been proposed to protect against stack smashing [11, 14]. Data execution prevention (DEP) prevents code from executing from pages allocated for stack or data [15], and

ISBN: 978-1-61804-336-8

$W \oplus X$ page protection schemes are now available at hardware level.

2.2 Return-oriented Programing

Protection mechanisms such as DEP are now commonly supported by major operating systems and make it impossible to perform code injection attacks. As a reaction, attackers devised mechanisms to bypass DEP. Techniques such as return-to-libc attacks, where the attacker subverts the control flow to call a function in the standard C library, can not be prevented by DEP because they execute valid code in memory segments. However, return-to-libc has its own weakness: it has clear attack code execution paths, since only regular library functions can be executed.

The CRA that allows arbitrary code execution was proposed recently. Return-oriented programming (ROP) [4] attacks are mounted as follows. The attacker identifies gadgets, which are sequences of instructions that end with a return instruction. Sufficient gadgets can be used to permit the composition of arbitrary attack code. Attackers inject a sequence of return addresses corresponding to a sequence of gadgets, using buffer overflow vulnerability. When returns are executed, the program counter is forwarded to the location of the first gadget. As that gadget terminates with a return, the return address is that of the next gadget, and so on. ROP executes instructions only from the code segment, so it is not prevented by DEP. A Turing-complete set of gadgets has been suggested on a number of architectures and operating systems [4].

Because of the seriousness of ROP attacks, a number of protection techniques have already been proposed. Davi et al. proposed the use of a reference monitor to detect the repeated execution of a small number of instruction sequences followed by a return [8]. Chen et al. monitor return properties to detect possible ROP attacks [5]. Li et al. propose rewriting binaries to eliminate the use of returns, completely preventing return-oriented attacks [12]. Most of the proposed techniques rely on validating the call-return behavior.

2.3 Jump-oriented Programming

The above solutions detect or prevent ROP attacks by monitoring the behavior of call and return instructions; however, a variation of the ROP attack was proposed. Instead of using call-return instructions, jump-oriented programming (JOP) [3] attacks employ branch instructions (jumps) to transfer control between the gadgets. Since no known defenses against ROP prevent JOP attacks, it is important to develop techniques that prevent JOP attacks with low overhead.

2.4 Control Flow Integrity

MVU is most related to recent work on using a reference monitor to track and enforce control flow integrity. In particular, Abadi et al. suggest that most root-kits and other malware initiate by changing the control flow graph (CFG) of the original program [1]. Thus, CFI tries to keep the program from these attacks by checking whether control flow corresponds with the original program CFG; this is the invariant that CFI tries to protect. If a branch that is not present in the CFG is encountered, the system quits the process and signals a CFI violation. ROP and JOP attacks are thwarted completely by enforcing full CFI at the branch level granularity. The CFI performance overhead is non-negligible, but moderate: an average of 16% performance penalty for SPEC 2000 benchmark.

In addition, CFI protection includes expensive static analysis, profiling, or deep binary analysis for the CFG construction. The existing software implementation of CFI also suffers a weakness on x86 and similar ISAs that feature variable instruction lengths: unintended control flow instructions can occur in the middle of multi-byte ISA instructions. These unintended instructions are not protected because no inline code can be inserted to check them.

2.5 Branch Target Buffer

Once an instruction is decoded as a branch, current processor initiate branch prediction to prefetch the next instruction. In doing so, past record of the corresponding branch is used for prediction. Branch target buffers (BTB) [9] is a cache unit that stores past target address of branches. If an instruction in fetch stage is identified as a branch, the instruction address is immediately sent to BTB. Then, the BTB checks if the corresponding entry exists. If so, it returns the value, which is the prediction of its target address.

In security perspective, entries in the BTB represent uncompromised addresses because every BTB entry is from the past target address that is already executed. If an attacker tried to run malicious code, the initial point of the attack would be different from the returned address from BTB. We can use this fact to narrow down the targets of verification.

2.6 Return Address Stack

In contrast to branches, past record do not predict call and return of subroutines with high probability, due to changing targets. Thus, modern speculative superscalar processors employ return address stack (RAS) for prediction, in which return address is pushed by

ISBN: 978-1-61804-336-8

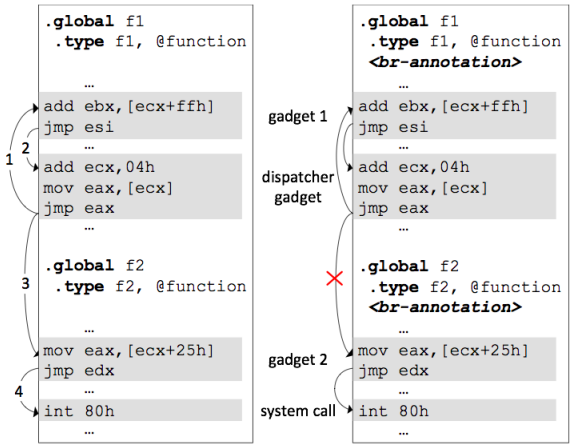

Figure 1: Impact of enforcing rules of MVU [10]. Left: possible JOP attacks using gadgets. Right: the protection by verification rule.

every subroutine call. When the subroutine ends by return instruction, the target of the return is fetched without delay, by popping an address off the stack.

RAS is a software transparent storage for return addresses, which can be used for validation. RAS can be employed for detecting malicious code since the code have never been executed. It also provides 98% or above prediction success rate in general.

3 Miss Verification Unit

3.1 Assumption and Threat Models

We assumed the safety of initial access of an address. Since CRAs require analysis of previously executed addresses and manipulate them, unexecuted code can hardly be a target of an attack. Another assumption is that the prediction will be incorrect under the CRA. To be specific, ROP or JOP compromises a program by programming counter manipulation and by making a gadget, and this changes corresponding target address. The CRA's nave alternation of target address can easily be detected by our scheme.

3.2 Rules of Miss Verification Unit

Speculating the behaviors of instructions that change next instruction addresses, we constructed Miss Verification Unit (MVU) that enforces verification to abnormal target instructions. The verification rules are composed of three restrictions that can prevent the arbitrary control flows. The following regulations for control flow checking are from Branch Regulation [10]:

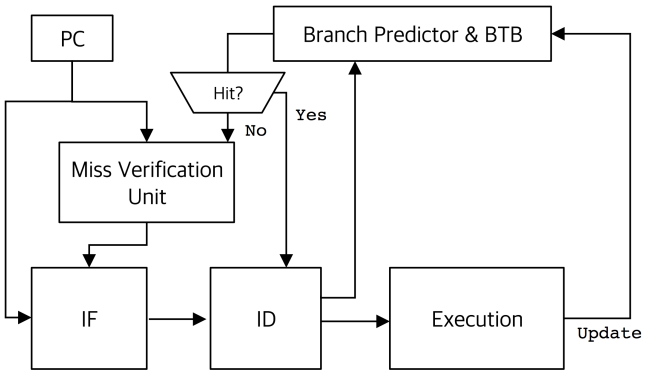

Figure 2: Architectual Layout of Miss Verification Unit. Instructions are verified when there is a miss. No extra verification is executed for other instructions.

The target address should be (1) an address in the same function; (2) the starting address in a different function; or (3) the return address of a corresponding call instruction. In case of $setjmp$ and $longjmp$, the instructions that jumps into the middle of function, are not regarded, since they are handled by operating systems. Thus, for the target address that fails to fit in these three categories, the execution will be blocked assuming that it is an illegal attempt.

Figure 1 shows a simple JOP attack scenario that uses one dispatcher gadget, two additional gadgets and a system call instruction (int 80h in x86)[10]. Suppose that the attacker is able to control writable memory. For this example, we assume that esi points to the dispatcher gadget, edx points to the system call instruction and ecx points to a memory location where addresses of gadgets 1 and 2 are stored contiguously. Furthermore, the parameters for the system call are assumed to be written in appropriate memory locations by the attacker. By enforcing above rules, the third hop is blocked since jmp in the function f1 is in the middle of an external function f2. This case does not match with any of our rules, and MVU will generate an exception.

3.3 Implementation Details of MVU

The type of instructions that governs the programming counter is typically branch (or jump), call, and return instructions.

For branches, a current Branch Target Buffer (BTB) holds all targets of executed branches. When a prediction miss occurs, either in case of an ordinary control flow change or an attack, BTB updates the corresponding entry. Return also has a call stack, which includes a collection of return addresses. We utilized Secure Call Stack (SCS) which can be accessed by only call and return instructions.

ISBN: 978-1-61804-336-8

Input	166.i	200.i	integrate.i	expr.i	scilab.i	average
CPI Baseline	1.5365	2.0938	1.5714	2.043	2.1022	1.86938
CPI with MVU	1.7308	2.0940	1.5714	2.043	2.1022	1.90824
Overhead	12.65%	0.02%	0.00%	0.00%	0.00%	2.08%

Table 1: The performance simulation result of MVU (with Simplescalar-3.0)

Also, there have been no architecture modules for function call prediction. We propose to construct Call Target Buffer (CTB), which stores the entry and end points of functions after execution. CTB will be acted as a checking table of function bound, since it stores the information of the starting and ending points of every function. Using CTB unit, the validity of branch target address can be easily checked.

When branch occurs, the instruction address is immediately sent to BTB after fetch stage. BTB provide the next PC if they have the corresponding entry, as shown in Figure. Normal execution continues if the prediction is correct, but a miss is generated if not. In the latter case, BTB miss handler sends the target value to MVU, which analyzes appropriateness in target address change. MVU accesses CTB to see its function bound. MVU enforces above regulations. If the address does not follow the regulations, MVU reports anomaly and flushes the pipeline. To the contrary, if the entry is not in the bound of CTB, it is regarded as an initial call to a new function; as well as the starting point of a function or any address in the function would be treated as legal. A jump inside a function is assumed to be invalid since set jump and long jump is handled by operating system. The instructions that direct control flows are need to be analyzed.

Employing MVU, more efficient protection can be delivered misprediction. We propose to show the diminished overhead by Simplescalar-3.0 benchmark. Also, the proportion of detected CRAs will be calculated to show the performance of MVU.

4 Evaluation

For evaluating the performance of MVU, we used Simplescalar-3.0 simulator. We simulated a 4-wide issue out-of-order core with 64KB L1 data and instruction caches, 512KB L2 cache and 2 MB L3 cache. Memory latency was assumed to be 100 cycles. Each benchmark was simulated for 1 billion committed instructions after fast-forwarding for the first 100 million instructions.

Table 1 shows the simulation result of MVU. We assumed that there will be 2 extra cycle for MVU execution, which is enough for simple checking sequence. The result shows that overhead of MVU in terms of CPI is 2.08% on average.

Due to the reliance on high performance of control flow prediction, MVU is shown to be a extremely efficient mechanism, compared to 16% overhead using CFI checking [1], and average 4% overhead in enforcing Branch Regulation [10]. Main contributor to this efficiency is pruning of the number of instruction to be verified. Nevertheless, the pruning was done to unnecessary parts, and does not decrease security effect.

Miss verification rules can effectively eliminate most of the potential gadgets and gadget dispatchers in ROP and its variants because it follows the same rules of the Brnach Regulation (BR) [9] (based on the assumptions and threat model of MVU in Section 3.1): The BR achieves 95% protection.

In terms of efficiency, MVU shows 2% overhead on average, as shown in Table 1. This significant reduction of performance overhead is mainly due to cuts in verification target. There is no miss from unconditional branches and conditional branches showed 96% prediction accuracy on average. In addition, the target is precisely aimed and verification is efficiently implemented, since the security unit is inserted in hardware level.

5 Conclusion

In this paper, we presented Miss Verification Unit (MVU), a novel light-weight defense mechanism against CRAs. MVU conducts verification of the instructions that are mispredicted by BTB, and disallows execution of these instructions unless their targets are within the same function or at the start of another function, with the exception of return statements that are matched to prior calls. Besides narrowing down the number of required verification, MVU makes construction of control flow graphs unnecessary, which saves time for static program analysis.

We demonstrated that the security benefits of BR are achieved at a very modest cost: about 2% performance loss, and simple hardware at the execution stage of the pipeline. By preserving these simple invariants, MVU showed effective prevention against the execution of malicious code. The perfor-

ISBN: 978-1-61804-336-8

mance and security test will be addressed through Simplescalar-3.0 benchmark.

This work was supported in part by the National Research Foundation (NRF 2015R1A2A01003242).

References:

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

[2] A. Baratloo, N. Singh, and T. Tsai. Transparent run-time defense against stack smashing attacks. *In Proceedings of the USENIX Annual Technical Conference*, 2000.

[3] T. Bletsch, X. Jiang, V. W. Freeh, and Z. Liang. Jump-oriented programming: a new class of code-reuse attack. *In Proceedings of the 7th ACM Symposium on Information, Computer and Communications Security*.

[4] E. Buchanan, R. Roemer, H. Shacham, and S. Savage. When good instructions go bad: generalizing return-oriented programming to risc. *In Proceedings of the 15th ACM conference on Computer and communications security*, page 2738, 2008.

[5] P. Chen, H. Xiao, X. Shen, X. Yin, B. Mao, and L. Xie. Drop: Detecting return-oriented programming malicious code. *In Proceedings of International Conference on Intelligent Sciences and Systems*, 2009.

[6] T. Chiueh and F.-H. Hsu. Rad: A compile-time solution to buffer overflow attacks. *In Proceedings of International Conference on Distributed Computing Systems*, 2001.

[7] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. *In Proceedings of the USENIX Annual Technical Conference*, 1998.

[8] L. Davi, A.-R. Sadeghi, and M. Winandy. Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. *In Proceedings of ACM Workshop on Scalable Trusted Computing*, 2009.

[9] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2012.

[10] M. Kayaalp, M. Ozsoy, N. Abu-Ghazaleh, and D. Ponomarev. Branch regulation: Low-overhead protection from code reuse attacks. *In Proceedings of the International Symposium on Computer Architecture*, 2012.

[11] R. B. Lee, D. Karig, J. P. McGregor, and Z. Shi. Enlisting hardware architecture to thwart malicious code injection. *In Proceedings of the International Conference on Security in Pervasive Computing*, page 237252, 2003.

[12] J. Li, Z. Wang, X. Jiang, M. Grace, and S. Bahram. Defeating return-oriented rootkits with return-less kernels. *In Proceedings of EuroSys*, 2010.

[13] Y. J. Park, Z. Zhang, and G. Lee. Microarchitectural protection against stack-based buffer overflow attacks. *Micro, IEEE*, 26(4):62–71, July 2006.

[14] S. Sinnadurai, Q. Zhao, and W. fai Wong. Transparent runtime shadow stack: Protection against malicious return modifications. http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.120.5702/, 2008.

[15] P. Team. Pax non-executable pages design & implementation. http://pax.grsecurity.net/docs/noexec.txt/, 2014.