Mitigating Code-Reuse Attacks on CISC Architectures in a Hardware Approach

Zhijiao Zhang, Yashuai Lü, Yu Chen, Yongqiang Lü, Yuanchun Shi

Mitigating Code-Reuse Attacks on CISC Architectures in a Hardware Approach

Zhijiao Zhang[1], YaShuai Lü[2], and Yu Chen[1]

[1]Department of Computer Science and Technology, Tsinghua University, Beijing, China
nudt_acer@163.com, yuchen@mail.tsinghua.edu.cn
[2]Academy of Equipment, Beijing, China
freelancer_lys@163.com

Abstract. Recently, code-reuse attack (CRA) is becoming the most prevalent attack vector which reuses fragments of existing code to make up malicious code. Recent studies show that CRAs especially jump-oriented programming (JOP) attacks are hard and costly to detect and protect from, especially on CISC processors. One reason for this is that the instructions of CISC architecture are of variable-length, and lots of unintended but legal instructions can be exploited by starting from in the middle of a legal instruction. This feature of CISC architectures makes the finding of so called *gadgets* for CRAs is much easier than that of RISC architectures. Most of previous studies for mitigating CRA on CISC processors rely on software-only means to tackle the unintended instruction problem, which makes their approaches either very costly or can only be applied under restricted conditions. In this paper, we propose two hardware supported techniques. The first, which is the main contribution of this paper, is to eliminate the execution of an unintended instruction. This technique only requires a few modifications to the processor and operating system. Furthermore, the proposed mechanism has little performance impact on the examined SPEC CPU 2006 benchmarks (-0.093% ~2.993%). Second, we propose using hardware control-flow locking as a complementary technique to our protection mechanism. By using the two techniques together, an attacker will have little chance to carry out CRAs on a CISC processor.

Keywords: CISC processor, Unintended instruction, Code-reuse attack, Instruction execution verification

1 Introduction

As the popularity of the Internet increases, so does the number of computer security threats from increasingly sophisticated attackers [1]. One common way to compromise a normal application is exploiting memory corruption vulnerabilities and transferring the normal program execution to a location under the control of the attacker. In these attacks, the first step is trying to overwrite a pointer in memory. Buffer overflow [2] and format string vulnerability exploitation [3] are two well-known techniques to achieve this goal. Once the attacker is able to hijack the control flow of the

application, the next step is to take control of the program execution to carry out some malicious activities. One of the typical and early attack techniques is code injection attack, in which a small payload that contains the machine code to perform the desired task is injected into the process memory. A wide range of solutions have been proposed to defend against memory corruption attacks, and to increase the complexity of performing buffer overflow and format string vulnerability exploitation [4-8]. To mitigate the code injection attack, a protection technique called W⊕X [9] was proposed. Under this protection regime, a memory page is either marked as writable or executable, but may not be both. Thus, an attacker may not inject data into a process's memory and then execute it simply by transfer control flow to that memory. Although the W⊕X technique is not foolproof [10, 11], it was thought to be a sufficiently strong protection regime that both the processor venders like Intel and AMD and prevalent operating systems like Windows [12], Linux [13], Mac OS X, and OpenBSD [14] now support it.

However, recently, attackers circumvented the W⊕X protection by employing *code-reuse* attacks (CRAs), which reuse the functionality provided by the exploited application. Using this technique, which was originally called return-to-libc [15], an attacker can compromise the stack and transfer the control to the beginning of an existing libc function. Often the system call system( ) is used to launch a process or mprotect( ) is used to create a writable, executable memory region to bypass W⊕X. In 2007, Shacham showed that W⊕X protection regime could be entirely evaded by so called return-oriented programming (ROP) [16] technique. In ROP, so called *gadgets* (small snippets of code ending in ret) are weaved together to achieve Turing complete computation without code injection. Since the advent of ROP, several effective defense techniques have been proposed [17-19]. However, a new class of code-reuse attacks called jump-oriented programming (JOP) that does not rely on rets has been proposed [20-22]. In these JOP attacks, the attacker chains the gadgets by using a sequence of indirect jump instructions, rather than **rets**, thus bypassing the defense mechanisms designed for ROP. Until now, there is no efficient technique can prevent both ROP and JOP attacks.

The x86 microprocessors are the most widely used general purpose processor series today. As a typical CISC architecture, instructions on the x86 platforms are of variable-length, and decoding an instruction from any byte offset is allowed. As a consequence, every x86 executable binary contains a vast number of *unintended* code sequences that can be accessed by jumping to an offset not on an original instruction boundary. Both the ROP and JOP take advantage of this feature to discover useful gadgets, which makes CRAs more easily to be carried out on CISC processors than that of RISC processors. Some of the previous studies that target for CRA defense tackle this problem in software-only approaches, which makes the applications of their techniques limited to constrained circumstances.

In this paper, we propose effectively mitigating CRAs on CISC architectures in a hardware supported approach. The main contributions of the work in this paper are:

- The unintended instruction problem of CISC processors is thoroughly resolved with only a few hardware and software modifications, and no application binary modification.
- The techniques are evaluated by cycle-accurate simulations of SPEC CPU 2006 benchmarks. The experimental results show that our proposed techniques have very little performance impact on these benchmarks.
- We further propose using hardware control-flow locking as a complementary technique to our protection mechanism.

The remainder of this paper is organized as follows. Section 2 describes the related work. The methodology and implementation details are presented in Section 3. The experimental evaluation is presented in Section 4. Section 5 offers some concluding remarks.

2 Related Work

The first step of a code reuse attack is to gain control of the program counter to divert program control flow to the first gadget. An attacker then may overwrite either the return address for the calling function or a function pointer with the address of the first gadget to divert the program control flow. Both software and hardware approaches were developed to prevent attackers from exploiting software vulnerabilities [4-8][23-26]. However, either these mechanisms are not adopted by hardware products or they are frequently not turned on as default when programs are compiled. As a result, attackers can always exploit software flaws to gain control of the program counter.

One tricky problem for CRA defense techniques on CISC platforms like x86 is the allowing of unintended instruction execution, which makes these defense techniques face more gadgets on CISC architectures. Before the advent of CRAs, some researchers of prior work [39, 40] realized that the unintended instructions could be a potential security problem, and solved it by imposing alignment in the environment of a sandbox. Since the advent of CRAs, previous CRA defenses rely on dynamic binary instrumentation tools monitoring unintended instructions without the help of hardware [34, 35], which limits their practical use.

Another problem is unintended control flow transfers. To solve this problem, a number of defense techniques have been proposed, and most of them are against ROP attacks. Several approaches use a shadow stack to prevent control flow manipulation that relies on overwritten stack values [27, 28]. Some try to detect gadget execution by monitoring return properties [17, 18], and others proposed monitoring pairs of call and return instructions [29, 30]. Several prevention approaches attempt to eliminate possible gadgets in library code [31, 32], and alternative strategies include creating binaries or kernels that lack necessary characteristics for ROP attacks [19, 33]. Most of these techniques rely on known characteristics of ROP attacks, and some were proved to have flaws that the defense mechanisms can be bypassed. Several approaches require recompilation of the program, library or kernel binaries, which may pose a problem when the source code is not available.

Since JOP is a recently proposed technique, there are only a few proposals that target for JOP defenses. Among these defense techniques, [34], [35] and [36] are pure software approaches. To distinguish normal program execution from CRA attacks, they rely on the software dynamic binary instrumentation. As a result, to apply their techniques, the programs must be executed through a dynamic binary instrumentation tool like Pin [37] or a virtual machine, which limits the practical use of their approaches. [30] and [38] are hardware supported approaches proposed by Kayaalp et al. By using binary rewriting, the branch regulation approach in [30] inserts markers to make jumps stay in a legal function. Branch regulation requires binary rewriting and cannot easily protect legacy binaries. It is also possible that a function may exist that can provide sufficient gadgets to mount an attack, and in this case security is not completely guaranteed. The work in [38] proposes a hardware supported signature-based protection mechanism. While this approach supports legacy software, it needs user to configure thresholds for CRA detection. Again, it is also one type of techniques that rely on known characteristics of CRAs.

3 Methodology

3.1 Elimination of Unintended Instructions

Figure 1 illustrates an example of unintended instruction sequence, which is similar to the one showed in [30]. The x86 assembly language used in this paper is written in Intel syntax. The disassembled code snippet is from the _IO_vfprintf function of libc-2.12-32. This code snippet consists of three instructions if decoded normally. However, if the decoding starts from the third byte of the call instruction, a different instruction sequence can be decoded as shown at the bottom of Figure 1, which contains three entirely different instruction. This unintended instruction sequence could possibly be used as a gadget, as the last jump instruction can be used to direct control flow to other gadgets (the memory location that register ebx points to). It should be noted that finding indirect jump instructions from unintended code sequence is extremely easy on x86 platforms, as byte FF represents the opcode of an indirect jump instruction on x86 [41], and FF is a common byte used in immediate values (bit-masks and sign bits of negative values).

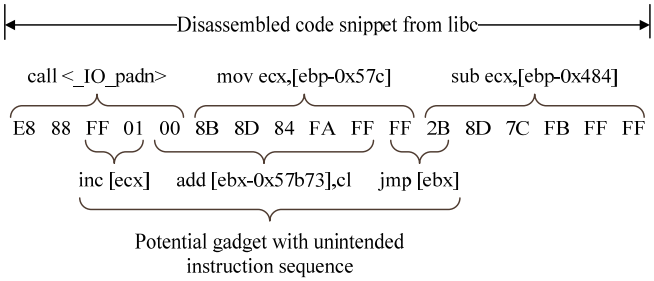

Fig. 1. Example unintended instruction sequence from libc.

Problem Analysis.

As well known, the processor pipeline can be roughly divided into five stages (i.e. IF, ID, EX, MEM and WB). Of the five stages, the instruction fetch stage fetches instructions from I-Cache according to the address pointed by program counter (PC), as illustrated in Figure 2. Therefore, it may be the most suitable stage to check whether an instruction is an intended instruction or not.

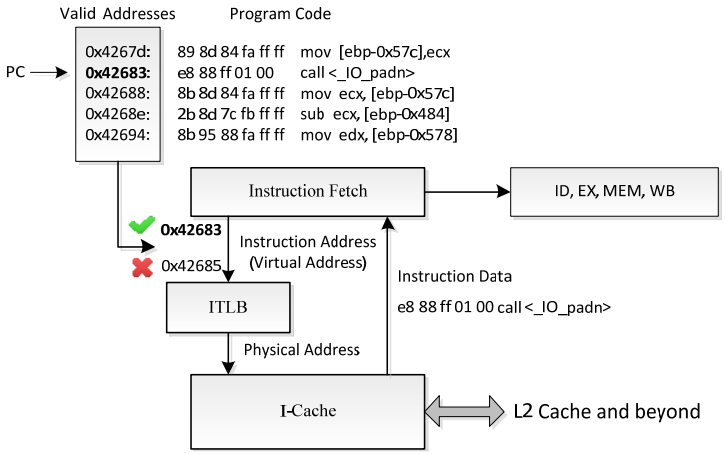

Fig. 2. Process of instruction fetch and instruction address validation.

For a variable-length instruction architecture, to know whether the current PC points to an unintended instruction, the processor needs a list of valid instruction addresses to be compared with, as illustrated in Figure 2. One way of acquiring the list of valid instruction addresses is disassembling the executable binary before its execution. When an executable binary has been loaded into memory, OS knows the location of each binary code section in the virtual memory space, and the valid address of each intended instruction can be obtained by sequentially disassembling the instructions from the start of each code section. Now with the list of valid instruction addresses, how to validate the legality of an instruction address? An inefficient preliminary solution is illustrated in Figure 3.

The solution depicted in Figure 3 first uses a recently validated address buffer (RVAB) to validate the current PC. The RVAB can be implemented in processor and functions similar to other LRU buffers in the processor. If the PC is not found in RVAB, then a further lookup will be needed. A program is often composed of several code sections, including the code sections from shared libraries. Instruction addresses within a code section are contiguous, but different code sections may not be contiguous in virtual memory space. Therefore, if a PC is not hit the RVAB, then the solution in Figure 3 first needs to find which code section this PC belongs to from a code section range list which can be derived from section information after the program has been loaded into memory. To save memory consumption, this preliminary solution organizes several contiguous addresses into an address chunk. The size of the chunk can be multiple of a byte. The first four bytes (for a 32-bit system) stores the first

address, and each following byte stores an offset from the first address. To find a specific address, the processor must search through the address chunks of a code section. If a binary search algorithm is used, the average time consumption can be O(log *n*) (*n* is the number of chunks in a code section), which means a processor may need accessing off-chip memory log *n* times to validate an address, and at the meantime, the processor pipeline must be stalled in order to wait for the result of the validation. Apparently, the overhead of this solution is too high for a modern processor.

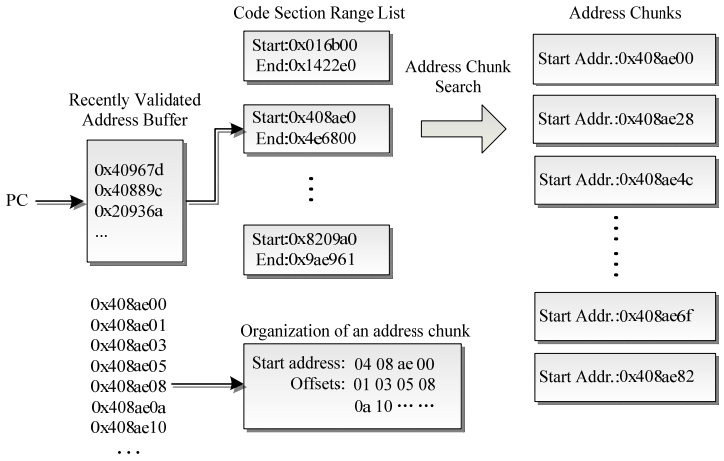

Fig. 3. A preliminary solution for instruction address validation.

Proposed Mechanism.

This subsection provides a more practical and efficient solution to the intended instruction validation problem in detail.

Recently Validated Address Buffer.

A proposed hardware implementation of RVAB is depicted in Figure 4. The buffer storage is divided into M sets and N lines. A branch target address to be validated is evaluated with a hash function to determine which set it belongs to. Each set contains N validated addresses or invalid empty entries. The address to be validated is compared with these addresses in parallel to quickly check whether the branch target hits the RVAB or not. For a practical implementation, a pseudo-LRU algorithm is used as the replacement strategy for the addresses within a set. Apparently, the hardware cost of RVAB mainly depends on its storage size. We will evaluate the choices of M and N in the experimental evaluation section.

Validation of Branch Target Address.

Theoretically, to ensure only intended instructions are executed, the addresses of all instructions that are going to be executed should be compared with valid instruction addresses. However, if we assume that the program starts from a valid instruction address, then we only need to validate the branch targets of control transfer instruc-

tions, as a CISC processor always decodes out instructions one by one sequentially. As a further optimization measure, if the W⊕X protection mechanism is present, we can restrict the validations to indirect control transfer instructions. This is based on the fact that the branch target of a direct control transfer instruction is written into the code section which cannot be modified when W⊕X protection mechanism is applied.

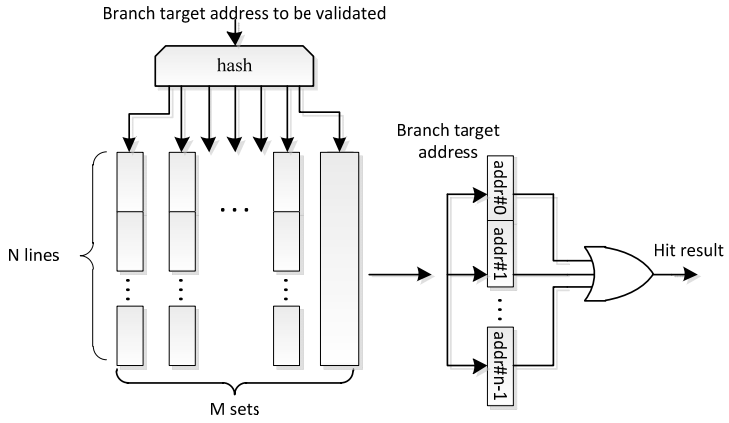

Fig. 4. The structure of recently validated address buffer.

Representation of Valid Instruction Addresses.

The solution in Figure 3 requires log*n* search times for validating an address that does not hit RVAB, which makes the searching very inefficient. Another way of representing instruction addresses is using one bit to indicate whether an address is the start of a valid instruction, Figure 5 depicts this idea.

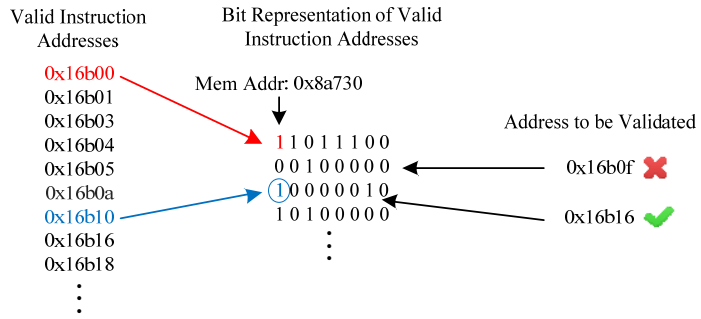

Fig. 5. Using one bit to indicate the validity of instruction address.

In Figure 5, suppose the instruction address validation data are stored beginning from 0x8a730 in memory, and the first bit represents the instruction address 0x16b00. If we want to know whether address 0x16b16 is a valid instruction address or not, then we can check the 7th bit of the byte at address 0x8a732 ((0x16b16 - 0x16b00)/8 + 0x8a730). Compared with the solution in Figure 3, this way of representing instruc-

tion addresses may consume more memory space (1/8 of the total code binary size), but greatly reduces instruction address validation time.

Code Section Range Lookup Table.

If the address representation in Figure 5 is used, then the code section range list acts more like a lookup table. The function of this code section range lookup table (CSRLT) is described in Figure 6. A CSRLT entry contains three items: the start and the end address of a code section and a memory pointer which points to the memory offset of the instruction address validation data of this code section. A full CSRLT is stored on off-chip memory and managed by OS. Like memory page TLB, some of the recently used entries in CSRLT are cached in an on-chip buffer (denoted by CSRLTB)

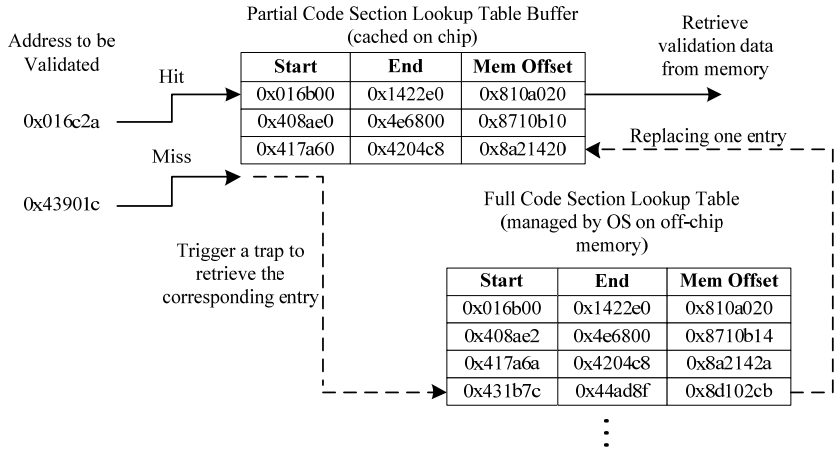

Fig. 6. The function of code section range lookup table.

Integration with Processor Pipeline.

Now we explain how to integrate our proposed mechanism with an out-of-order processor pipeline, which is illustrated in Figure 7. As there is no need to validate every fetched instruction's address except for branch target, the validation process is better moved from instruction fetch stage to commit stage. There are two reasons behind this: firstly, the real branch target of a control transfer instruction can only be determined after the EX stage; secondly, some of the control instructions in EX stage may be aborted due to failed speculation and commit stage is where precise exceptions are usually carried out in out-of-order processors. However, validating the branch target address when the instruction is just about to commit will stall the processor pipeline for each validation even if the address hits RVAB. Thus, a better solution is carrying out the validation when the control instructions are at the front of the ROB committing queue but have not reached the head yet, and using one bit to indicate the branch target is valid or not.

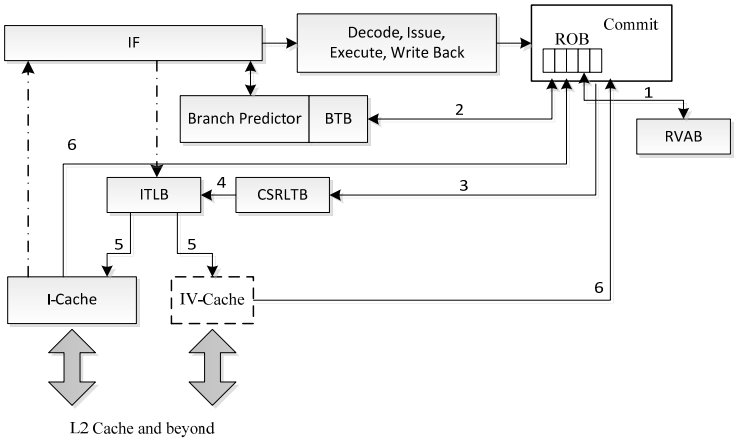

Fig. 7. The integration with an out-of-order pipeline.

3.2 Further Protection Measures

Statistical results in [30] shows that limiting the dispatcher gadgets to intended instructions greatly reduces the number of potential dispatcher candidates. However, the remaining dispatcher candidates can still be expored for carrying out JOP attacks. As a result, we propose using *control flow locking* (CFL) as the complementary technique to our mechanism. CFL was proposed by Bletsch et al. in [36].

The idea of CFL is illustrated in Figure 8. This code snippet is from **libssl.a**. The actual destination of indirect jump in address 0x451 is address 0x4f0. In [36], to implement CFL, a small snippet of *lock* code is inserted before each indirect control flow transfer. This code asserts the *lock* by simply changing a certain *lock value* in memory, and each valid destination for that control transfer contains the corresponding unlock code, which will de-assert the *lock* if and only if the current *lock value* is deemed "valid". In Figure 8, variable *k* is the *control flow key*, and value 1 means locked while 0 means unlocked.

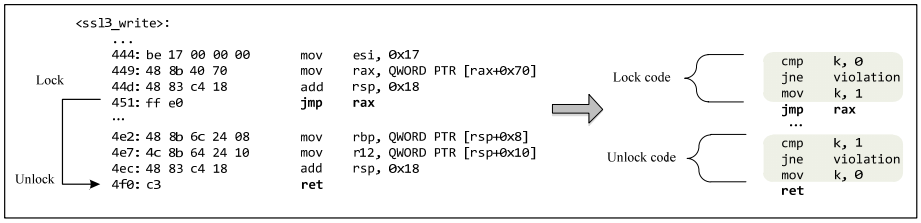

Fig. 8. An illustrative example of CFL.

As this paper focuses on hardware supported techniques to mitigate JOP attacks, we propose using hardware to efficiently implement a weaker version of CFL. First, for each committed indirect jump, the processor needs to record that the processor is in a *jump locking* state. Second, we can devise a new instruction or just add a prefix to

current instructions to denote an indirect jump destination. The role of the new instruction or prefix is to *unlock* the *jump locking* state, and this *unlocking* instruction must be committed just after an indirect jump instruction. Apparently, this mechanism is not hard to implement in hardware. For the hardware CFL, the compiler is responsible for correctly inserting indirect jump destination instruction or prefix when translating high level language source code into machine binary code. A difference to software CFL is that we omit the *lock value*. We argue that implementing *lock value* will introduce much more hardware and software cost. With the elimination of unintended instructions and the support of hardware CFL, it is already extremely hard to find out enough gadgets to carry a JOP attack.

4 Performance Evaluation

For evaluating the performance of our techniques, we used the gem5 [43] simulator to simulate a 4-core x86 CMP. All our experiments were carried out on a high-end desktop computer which has a 3.4 GHz Core i7-4770 CPU with 16GB memory, and the operating system is CentOS 6.4 x86-64. We selected 21 benchmarks from the SPEC CPU2006 [44] benchmark suit and used test input data set for our experiments. These benchmarks were compiled using CentOS's native GCC compiler (version 4.4.7) with -O3 optimizations. For each benchmark, we ran the simulation for 10 billion instructions or until its completion.

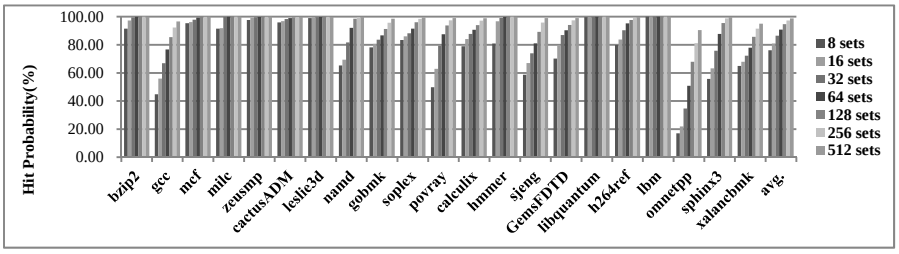

Fig. 9. RVAB hit probability for validations of all control instructions.

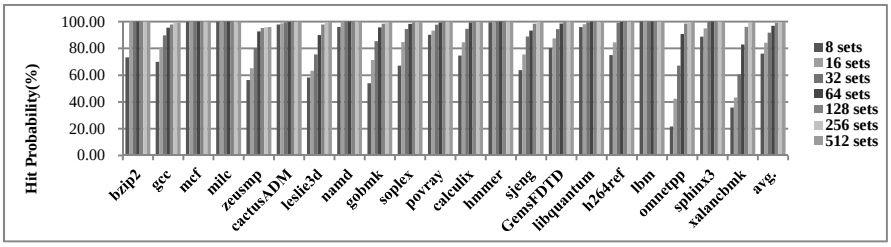

Fig. 10. RVAB hit probability for validations of indirect control instructions.

First, we will evaluate the configurations of RVAB. In the work of this paper, we used 4 entries for each set, and pseudo-LRU for replacement strategy. For the hash

function, we simply used lower part of the address as set index. Now we only need to investigate how many sets will be enough for a high RVAB hit probability.

Figure 9 illustrates the RVAB hit probability when all control instructions were considered, and Figure 10 illustrates the result when only indirect control instructions were validated. It can be observed that for the same number of set and the same benchmark, the hit probability of Figure 10 is higher than that of Figure 9. An important observation is that when the number of set reaches 128, the hit probability does not change very much, especially for 256 sets and 512 sets. For the benchmarks in Figure 9, the average hit probabilities of 128 sets, 256 sets and 512 sets are 94.68%, 97.35% and 98.84%; and for the benchmarks in Figure 10, the average hit probabilities of 128 sets, 256 sets and 512 sets are 99.11%, 99.61% and 99.70%. Based on this observation, we selected 128 as the set number for the following experiments, and the total number of RVAB entries is 128x4 = 512. As we can see, the hardware cost of RVAB is very small.

We evaluated three configurations for performance impact evaluation. The first configuration is validating all control instructions and using branch predictor as lookup backup (denoted as BP & AC); the second is only validating indirect control instructions and using BP as lookup backup (denoted as BP & IC); the third configuration is only validating indirect control instructions but without using BP as backup. Figure 11a shows the IPC results of the baseline and the three configurations. Here we count x86 micro-ops as instruction counts. To get a clearer view, Figure 11b-11d shows relative performance slowdown of these three configurations.

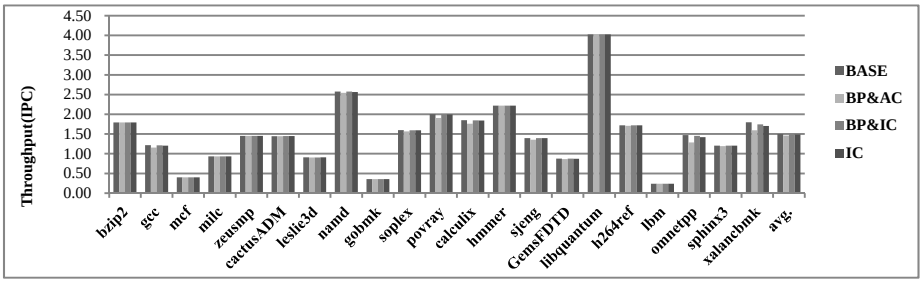

Fig. 11. a. Throughput comparison (IPC) of baseline and three different configurations.

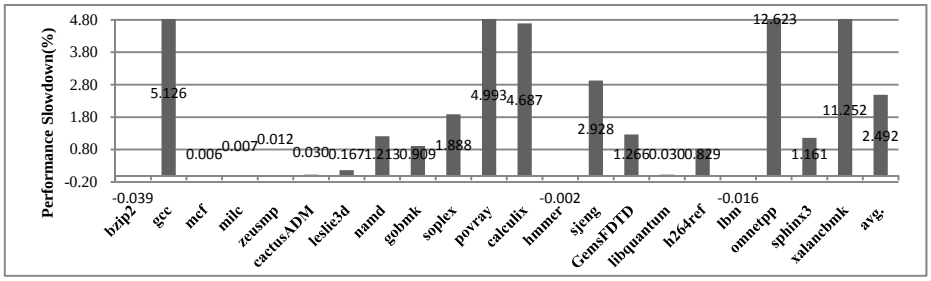

Fig. 11. b. Relative performance slowdown of BP&AC configuration.

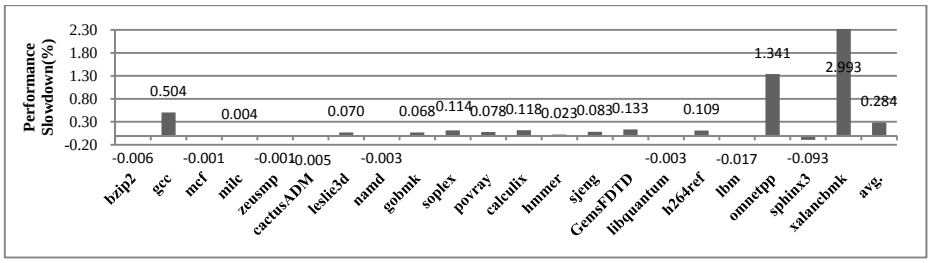

Fig. 11. c. Relative performance slowdown of BP&IC configuration.

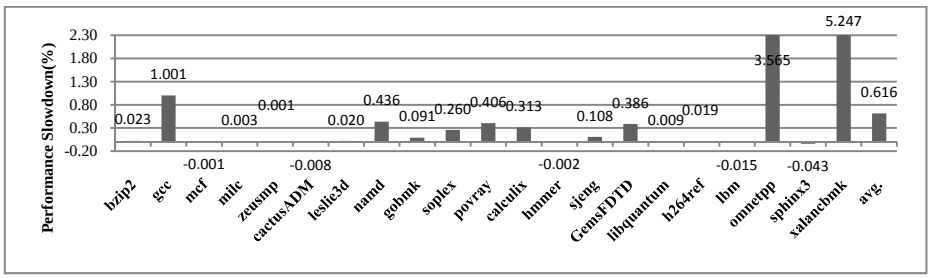

Fig. 11. d. Relative performance slowdown of IC configuration.

From Figure 11a, we can see that the performance impacts of our proposed methodology are very small for all three configurations. For the BP&AC configuration, the average performance slowdown is 2.492%, the highest performance slowdown is 12.623% (**omnetpp**), the smallest is -0.039%(**bzip2**, which means IPC gets a little increase), and there are 11 benchmarks whose performance slowdowns are below 1%. As we expect, the BP&IC configuration gets the best performance result. Its performance slowdown is between -0.093% and 2.993%, and is 0.284% on average. Only two benchmarks' slowdowns exceed 1%. Without BP as backup, the performance result of the IC configuration is a little worse than the BP&IC configuration, but better than the BP&AC configuration. Its slowdown is between -0.043% and 5.247%, and is 0.616% on average.

Since the W⊕X mechanism is widely used in modern systems, only validating indirect control instructions is safe enough. The experimental data in Figure 11 indicate that this approach has very little impact on program performance. Even without the help of BP, the highest performance slowdown of IC configuration is only 5.247%, and only three benchmarks have slowdowns above 1%.

5 Concluding Remarks

In this paper, we proposed hardware supported techniques to mitigate CRAs on CISC architectures. We addressed the CRA problem by two protection mechanisms: (1) preventing executions of unintended instructions; (2) preventing unintended control flow transfers. The work in this paper mainly focuses on the first problem, as no pre-

vious work addressed it by an effective hardware approach. We addressed the second problem by using a hardware version of CFL. We also demonstrated that our approach has a very modest cost on both hardware and software. For the BP&IC configuration, the performance loss is -0.093% ~2.993%.

Acknowledgements

This work is supported in part by the Natural Science Foundation of China under Grant No. 61170050, National Science and Technology Major Project of China （2012ZX01039-004）. The authors would also like to thank anonymous reviewers who have helped us to improve the quality of this paper.

References

1. Symantec: Internet Security Threat Report 2014, http://www.symantec.com/security_response/publications/threatreport.jsp
2. One, A.: Smashing the stack for fun and profit. J. Phrack magazine, 7(49), 14–16 (1996)
3. Scut, T. T.: Exploiting format string vulnerabilities (2001)
4. Cowan, C., Beattie, S., Johansen, J., and Wagle, P.: August. Pointguard TM: protecting pointers from buffer overflow vulnerabilities. In *Proceedings of the 12th conference on USENIX Security Symposium*. Vol. 12, 91–104 (2003)
5. Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., ... and Hinton, H: StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Usenix Security* 98, 63–78 (1998)
6. Cowan, C., Wagle, P., Pu, C., Beattie, S., and Walpole, J.: Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings of DARPA Information Survivability Conference and Exposition.* DISCEX'00. IEEE. Vol. 2, 119–129 (2000)
7. Etoh, H., and Yoda, K.: GCC extension for protecting applications from stack-smashing attacks, 2014, http://www.research.ibm.com/trl/projects/security/ssp/
8. Shield, S.: A stack smashing technique protection tool for Linux, 2014, http://www.angelfire.com/sk/stackshield/
9. Pax Team: Non-executable pages design and implementation, http://pax. grsecurity.net/docs/ pageexec.txt
10. Krahmer, S.: x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique, 2005, http://www.suse.de/krahmer/no-nx.pdf
11. McDonald, J.: Defeating Solaris/SPARC non-executable stack protection. Bugtraq (1999)
12. Microsoft. KB 875352: A detailed description of the Data Execution Prevention (DEP) feature in Windows XP Service Pack 2, Windows XP Tablet PC Edition 2005, and Windows Server 2003, Sept. 2006, http://support.microsoft.com/KB/875352
13. Designer, S. Linux kernel patch from the Openwall project, http://www.openwall.com/linux.
14. OpenBSD Foundation. OpenBSD 3.3 release. 2003, http://www.openbsd.org/33.html
15. Solar Designer.: Return-to-libc attack. *Technical report*, bugtraq. (1997)

16. Shacham, H.: 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*. ACM, 552-561 (2007)
17. Davi, L., Sadeghi, A. R., and Winandy, M.: Dynamic integrity measurement and attestation: towards defense against return-oriented programming attacks. In *Proceedings of the 2009 ACM workshop on Scalable trusted computing*, 49–54 (2009)
18. Chen, P., Xiao, H., Shen, X., Yin, X., Mao, B., and Xie, L.: DROP: Detecting return-oriented programming malicious code. In *Information Systems Security*. Springer Berlin Heidelberg, 163–177 (2009)
19. Li, J., Wang, Z., Jiang, X., Grace, M., and Bahram, S.: Defeating return-oriented rootkits with return-less kernels. In *Proceedings of the 5th European conference on Computer systems*, ACM, 195–208 (2010)
20. Bletsch, T., Jiang, X., Freeh, V. W., and Liang, Z.: Jump-oriented programming: a new class of code-reuse attack. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM. 30–40 (2011)
21. Checkoway, S., Davi, L., Dmitrienko, A., Sadeghi, A. R., Shacham, H., and Winandy, M.: Return-oriented programming without returns. In *Proceedings of the 17th ACM conference on Computer and communications security*. ACM. 559–572 (2010)
22. Chen, P., Xing, X., Mao, B., Xie, L., Shen, X., and Yin, X.: Automatic construction of jump-oriented programming shellcode (on the x86). In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM. 20–29 (2011)
23. McGregor, J. P., Karig, D. K., Shi, Z., and Lee, R. B.: A processor architecture defense against buffer overflow attacks. In *Proceedings of Information Technology: Research and Education.*. ITRE'03. International Conference. 243–250 (2003)
24. Lee, R. B., Karig, D. K., McGregor, J. P., and Shi, Z.: Enlisting hardware architecture to thwart malicious code injection. In *Security in Pervasive Computing*. Springer Berlin Heidelberg. 237–252 (2004)
25. Davi, L., Sadeghi, A. R., and Winandy, M.: ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ACM. 40–51 (2011)
26. Xu, J., Kalbarczyk, Z., Patel, S., and Iyer, R. K.: Architecture support for defending against buffer overflow attacks. In *Workshop on Evaluating and Architecting Systems for Dependability* (2002)
27. Davi, L., Sadeghi, A. R., and Winandy, M.: ROPdefender: A detection tool to defend against return-oriented programming attacks. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*. ACM. 40–51 (2011)
28. Francillon, A., Perito, D., and Castelluccia, C.: Defending embedded systems against control flow attacks. In *Proceedings of the first ACM workshop on Secure execution of untrusted code*. ACM. 19–26 (2009)
29. Chen, P., Xing, X., Han, H., Mao, and B., Xie, L.: Efficient Detection of the Return-Oriented Programming Malicious Code. In: *International Conference on Information Systems Security (ICISS)* (2010)
30. Kayaalp, M., Ozsoy, M., Abu-Ghazaleh, N., and Ponomarev, D.: Branch regulation: Low-overhead protection from code reuse attacks. In: *International Symposium on Computer Architecture (ISCA)* (2012)
31. Hiser, J., Nguyen-Tuong, A., Co, M., Hall, M., and Davidson, J. W. ILR: Where'd My Gadgets Go? In *IEEE Symposium on Security and Privacy*. IEEE. 571–585 (2012)

32. Pappas, V., Polychronakis, M., and Keromytis, A. D.: Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *Security and Privacy (SP)*, IEEE Symposium. 601–615 (2012)
33. Onarlioglu, K., Bilge, L., Lanzi, A., Balzarotti, D., and Kirda, E.: G-Free: defeating return-oriented programming through gadget-less binaries. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC)*. ACM. 49–58 (2010)
34. Huang, Z., Zheng, T., Shi, Y., and Li, A.: A Dynamic Detection Method against ROP and JOP. In: *International Conference on Systems and Informatics (ICSAI)* (2012)
35. Jacobson, E. R., Bernat, A. R., Williams, W. R., and Miller, B. P.: Detecting Code Reuse Attacks with a Model of Conformant Program Execution. In *Engineering Secure Software and Systems*. ESSoS'14. Springer International Publishing. 1–18 (2014)
36. Bletsch, T., Jiang, X., and Freeh, V. Mitigating code-reuse attacks with control-flow locking. In *Proceedings of the 27th Annual Computer Security Applications Conference*. ACM. 353–362 (2011)
37. University of Virginia, Pin, `http://www.cs.virginia.edu/kim/publicity/pin`
38. Kayaalp, M., Schmitt, T., Nomani, J., Ponomarev, D., and Abu-Ghazaleh, N.: SCRAP: Architecture for signature-based protection from code reuse attacks. In *High Performance Computer Architecture* (HPCA). IEEE 19th International Symposium. 258–269, 23–27 (2013)
39. McCamant, S., and Morrisett, G.: Efficient, verifiable binary sandboxing for a CISC architecture. In *MIT Technical Report*. MIT-CSAIL-TR-2005-030 (2005)
40. Yee, B., Sehr, D., Dardyk, G., Chen, J. B., Muth, R., Ormandy, T., ... and Fullagar, N.: Native client: A sandbox for portable, untrusted x86 native code. J. Security and Privacy, 30th IEEE Symposium. 53(1):79–93 (2009)
41. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual, vol. 2 (2013)
42. Udis86 Disassembler Library for x86/x86-64, `http://udis86.sourceforge.net/`
43. Binkert, N., Beckmann, B., Black, G., Reinhardt, S.K., Saidi, A., Basu, A.,… and Wood, D.A.: The gem5 simulator. J. Computer Architecture News. 39:1–7 (2011)
44. HENNING, J. L.: Spec cpu2006 benchmark descriptions. J. ACM SIGARCH Computer Architecture News, 1–17 (2006)