Detecting Code Reuse Attacks with Branch Prediction

Yongsuk Lee and Gyungho Lee, Korea University

Code reuse attacks (CRAs) allow attackers to produce malicious results by using legitimate code binaries in memory. The authors propose incorporating control-flow validation into the processor's instruction execution pipeline, along with a mis-prediction validation unit and a branch prediction unit, to help identify attacks.

Computers are exposed to constant threats from software attacks. Researchers have explored many different ways for ensuring trustworthy software behavior, yet attacks continue to become more sophisticated and often outpace the protections. There have been numerous tactics that successfully block compromised program control-flows based on specific attributes of the attack itself; however, these have led attackers to launch new, more sophisticated types of attacks, which bypass the protections.

A code injection attack attempts to inject malicious code into memory.[1] This strategy is less often used due to new, effective protection mechanisms such as DEP (data execution prevention), which prevents code injected into the data area from starting its execution. Attackers have responded to this advance in protection methods with code reuse attacks (CRAs). CRAs can perform malicious actions by reusing the existing code binaries already in memory. For example, the return-to-libc attack modifies a return address saved in the memory so the program will "call" an existing C library function, instead of returning to the caller. A more evolved example is return-oriented programming (ROP) and its variants.[2-5] The attack forms "gadgets," each of which is a short sequence of instructions that ends with an indirect branch instruction such as call, return, and jump with register, from legitimate code already in memory. Then the attack payload is a sequence of addresses for the gadgets to compose a desired functionality. A sufficient number of gadgets can form an attack program with arbitrary functionality ("Turing complete").

Maintaining control-flow integrity (CFI), that is, ensuring at runtime that the program control-flow is following the control-flow information implied at the machine instruction level in the program, can be a basic principle for providing sound protection against most CRAs.[6,7] CFI implementations instrument the program to validate the control-flow transfer of an indirect branch instruction instance using a reference control-flow graph (CFG) generated statically or dynamically from the program. Ensuring that each indirect branch instance follows the CFG involves validating the target address for the given indirect branch instance.

A CFG can be viewed as a table with entries consisting of an indirect branch instruction location and its target address, called an indirect branch pair (IBP). Following the view, it is possible to see the similarity between the CFI per the CFG and the branch prediction found in almost all modern processors; the branch prediction verifies the predicted target address for a branch instruction instance. One key observation is that the branch prediction, especially for indirect branch instructions, is already doing control-flow validation: the target addresses stored in the branch prediction unit (BPU) provide recently encountered and validated IBPs.

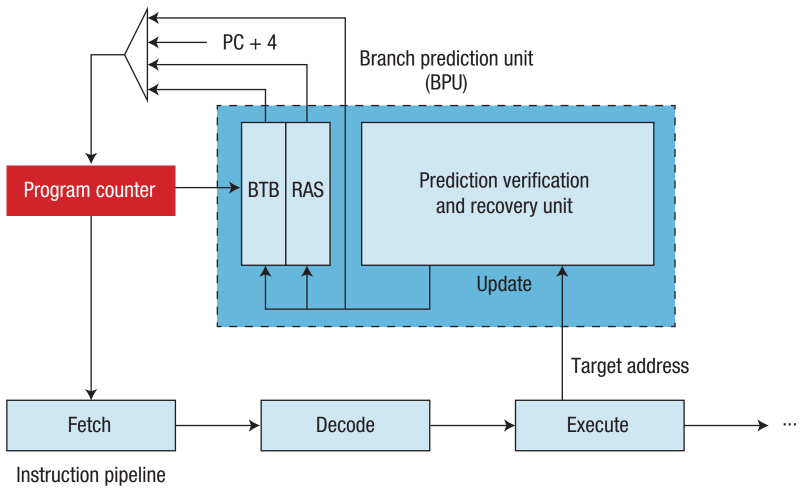

FIGURE 1. Branch prediction allows the processor to move on with the predicted target address from the branch target buffer (BTB) or return address stack (RAS). If the predicted target address turns out to be incorrect later in the instruction pipeline, the processor re-fetches the instruction with the correct target address (available usually at the execute stage of the branch instruction) and nullifies the instructions fetched with the predicted target address.

We propose incorporating the control-flow validation into the processor's instruction execution pipeline. Instead of adding instrumentation for CFI implementation, control-flow validation becomes an integral part of the indirect branch instruction execution, which will provide tighter protection and, at the same time, less performance overhead. This article introduces a mis-prediction validation unit (MVU) alongside the BPU for control-flow validation. An MVU validates the mis-predicted target address, whether it is caused by a legitimate target address for a legitimate control-flow or a compromised target address from an attack. Since an MVU works in tandem with a BPU, it allows us to narrow the control-flow validation scope within the mis-predicted branches and to incorporate an MVU into the instruction execution pipeline with little overhead. An MVU allows control-flow validation to be integrated into the processor's instruction execution pipeline like a memory management unit (MMU) enabling the virtual-to-physical address translation to be an integrated part of the instruction execution pipeline.

CODE REUSE ATTACKS AND BRANCH PREDICTION

When fetching an instruction for execution, modern pipelined processors use branch prediction to fetch the next instruction without waiting for the target address of the branch instruction. In doing so, the BPU stores a record of previous target addresses. For example, the

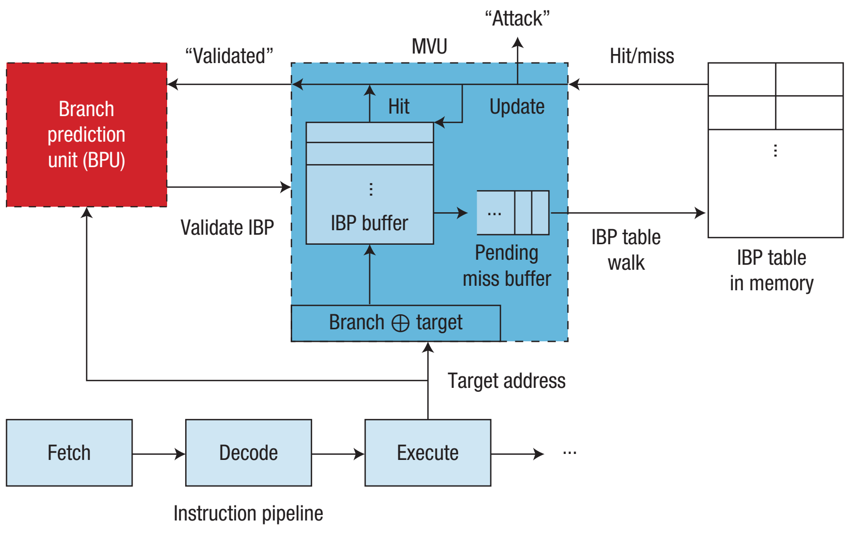

FIGURE 2. Mis-prediction validation unit (MVU). While the branch prediction unit (BPU) verifies whether the predicted target is correct, the MVU validates the indirect branch pair (IBP) of the mis-predicted indirect branch instruction to be sure it is not compromised.

last encountered target address of a branch instruction is stored in a small buffer called the branch target buffer (BTB), and the return address for the last call is stored in a fast hardware stack, called the return address stack (RAS).[8] For the current program counter (PC) value, the BPU checks if the corresponding entry exists in the record of previous target addresses. If so, the target address found in the record becomes the predicted target address. Otherwise, the next physical instruction is predicted.

The processor uses the predicted address to fetch the next instruction for execution without waiting for the branch instruction to provide the correct target address. While the processor moves on with the predicted target address, the BPU waits to verify if the predicted target is correct (branch prediction verification) until the target address from executing the branch instruction becomes available, which is usually at the execute stage of the processor's instruction pipeline. If the predicted target address turns out to be incorrect, that is, it is a mis-prediction, the processor rolls back the PC and re-fetches the instruction with the correct target address (mis-prediction recovery) (see Figure 1).

With no control-flow compromises, a mis-prediction occurs if an indirect branch instruction instance takes a different target address, out of several legitimate target addresses, from the previous instances. Under a CRA, a branch mis-prediction occurs when the control-flow transfer is hijacked to an unexpected position in the existing code because it is not a part of the program's legitimate control-flow and has not been previously executed. A branch target used by the attack might be legitimate, but it causes a branch mis-prediction because the target address is not the one that results from a normal execution of the branch instruction at the location. Alternatively, the branch location could be legitimate while the target address is not. In either case, the IBP is not legitimate. Note that from the control-flow validation perspective, the target addresses from branch prediction represent uncompromised addresses, because every past target address is already validated if the validation process is in place. We can use this fact to narrow the validation scope for detecting CRAs down to mis-predicted indirect branch instances.

MIS-PREDICTION VALIDATION UNIT

To detect a CRA, the mis-prediction validation unit (MVU), along with the branch prediction (see Figure 2), distinguishes whether the cause for a branch mis-prediction is from a legitimate control-flow transfer or from a compromised control-flow transfer by checking the reference CFG.

Validating Branch Mis-prediction

To see how a CRA changes the indirect branch pair (IBP)—that is, the information on the branch location and its target address—let us consider a return-to-libc attack as an example. A return-to-libc attack compromises the return address saved in the runtime stack to cause an illegitimate control-flow to a C library function. When the PC points to the return instruction, the processor fetches its target instruction with the predicted return address from the RAS without waiting to see the return address saved in the runtime stack. When the return address from the runtime stack becomes available later as the return instruction is executed, the prediction verification compares it with the predicted one from the RAS.

Under the attack, the return address from the runtime stack in memory is the one compromised, while the one from the RAS is not. The branch verification concludes it is a mis-prediction because the return address from the runtime stack and the predicted one do not concur. Unfortunately, when the recovery process for the branch mis-prediction rolls back the processor's progress with the predicted return address and re-fetches the instructions, it uses the compromised return address from the runtime stack. To avoid reverting the legitimate flow to the compromised flow of the attack, an MVU validates the IBP for the mis-predicted branch, that is, whether the mis-prediction is due to a compromised control-flow from the attack.

This paper assumes that the IBP table—namely, the set of IBPs collected from the profiled traces—is the reference CFG in memory. One might also generate the IBP table statically as in most CFI implementations; however, we have used the profiled traces for ease of experimentation, including obtaining the IBP information for dynamically linked routines. We assume that the IBP table is loaded along with the code binary.

Note that each IBP provides a unique identifier for each indirect branch with each of its target addresses. Assuming the code text area in memory is read only, the IBP set does not include direct branch instructions including conditional branches, because their target addresses are fixed and described as a part of the instruction bit pattern itself. The IBP table makes an ideal fine-grained CFG in terms of identifying each indirect branch instance uniquely.[6,7,9]

For IBP table access, we provide a small cache, called the IBP buffer. Instead of loading the IBP table into the data cache, a separate software transparent buffer provides the isolation to secure the IBP information. Note that the IBP buffer is not read or written directly by software. It is read and updated by the MVU as an integral part of the indirect branch instruction execution. When the BPU goes through the branch verification and recovery, the MVU validates the IBP for the mis-predicted branch instance by accessing the IBP buffer (see Figure 3). If the MVU finds the IBP in the IBP buffer, it needs no action. As long as the IBP buffer access takes the same or less time for the branch prediction verification and recovery, the control-flow validation by the MVU is integrated into the existing instruction pipeline with no extra delay.

Since the MVU can only start checking the IBP buffer after the target address from the branch instruction is available, one might think that the MVU's access to the IBP buffer will add a delay to the instruction execution pipeline. When the target address is available from the branch instruction, it is provided to the BPU for prediction verification. If it turns out to be a mis-prediction, the recovery process rolls back the instruction fetches with the verified target address. This verification and recovery process adds a mis-prediction penalty of a few cycles to the processor's instruction pipeline. Since the target address from the branch instruction is available to the MVU at the same time as to the BPU, an MVU can finish the validation early enough to not cause additional delay if it finds the IBP in the IBP buffer.

If the MVU experiences an IBP buffer miss, it needs to access the IBP table in memory (IBP table walk). The MVU needs to signal an attack if the IBP table walk fails: an IBP table search failure means the IBP is not legitimate, that is, there is a compromised control-flow

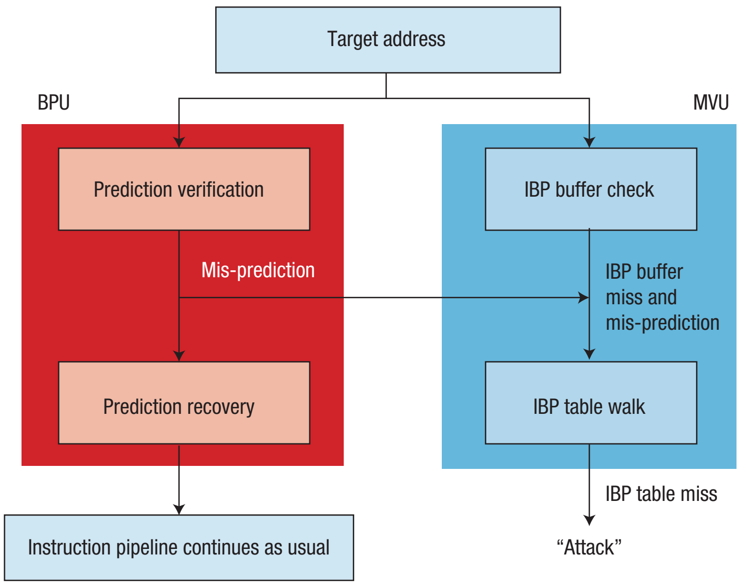

FIGURE 3. Overall mis-prediction validation flow.

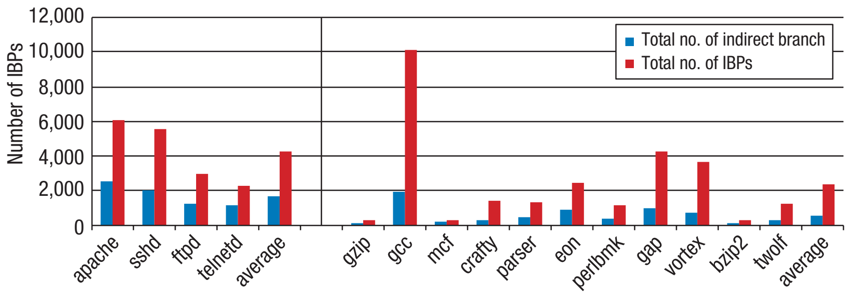

FIGURE 4: Number of IBPs. Indirect branch profiling for SPEC2000 CPUint and four server programs: apache, sshd, ftpd, and telnetd.

from the attack. Since the IBP table walk may take a long time because of the memory access, it will take longer than the BPU's branch verification and recovery. However, the IBP table walk for handling an IBP buffer miss does not need to interfere with the program execution; the instruction pipeline can move on without waiting for the MVU's validation result.

In normal situations, namely, when the control-flow is not compromised, the MVU does not interfere with the program execution. With a compromised IBP, the program execution needs to be blocked anyway as the MVU's IBP table walk failure prevents the attack from progressing. The MVU's control-flow validation can be decoupled from the instruction pipeline by introducing a small buffer to hold incoming mis-predicted indirect branches when an IBP table walk is pending (see "pending miss buffer" in Figure 2). Without the buffer, an MVU would need to stall the instruction pipeline until the IBP table walk finishes, which introduces a performance penalty.

IBP Buffer

To see how large the IBP table might become, we collected IBPs from SPEC2000 CPUint benchmark programs and also from four popular server programs: apache, sshd, ftpd, and telnetd. For the server programs, we collected IBPs from actual daily use and also with synthetic input data. We ran the SPEC benchmark programs with the provided regular input datasets from start to finish without any skipping. We used the Intel VTune Analyzer to collect the IBPs. We found the indirect branch instructions to be about 1.5 percent of all the instructions executed.

The profiling results show that the IBP tables are quite modest in size: the IBP table size for all the programs except *gcc* is less than 6,000, whereas it is slightly above 10,000 IBP's for *gcc* (see Figure 4). Although we know from the programs used in our experiments that IBP table size is usually small, it is still desirable to have a cache for the IBP tables to assist the MVU to validate branch mis-predictions with a shorter delay than the BPU's mis-prediction penalty.

Multiple target addresses are possible for a given indirect branch instruction. Although it varies from program to program, each indirect branch in the programs we tested has two to five different targets on average (see Figure 3). To reduce the IBP buffer miss rate, it is desirable to allow multiple IBPs to be stored in the buffer for a given PC value at the same time. Instead of indexing the IBP buffer via PC only, doing an exclusive-or (XOR) of the branch address and its target address will distribute its IBPs over the buffer address space. We might want to improve the IBP buffer hit rate with a set-associative structure to store multiple target addresses for each indirect branch in a more explicit way. However, we did not explore further IBP buffer refinement because even a small buffer can provide a very high hit rate with the simple XOR indexing.

PERFORMANCE OVERHEAD

To evaluate the potential performance overhead from the MVU's control-flow validation, we studied SPEC2000 CPUint benchmark programs with a Simplescalar-3.0 simulator, simulating a 4-wide issue out-of-order 9-stage pipeline core with 64 KB L1 data and instruction caches. We simulated each benchmark for 1 billion committed instructions after fast-forwarding for the first 100 million instructions. We used the following assumptions for the evaluation environment, the processor, and the MVU.

Processor:

- Pipeline—4-issue, 9-stage
- Issue queue size—16
- Reorder buffer size—64
- Branch predictor—gshare predictor with 4,096 counters and 16-entry RAS
- Mis-prediction penalty—5 cycles
- L1 caches—64 KB instructions/64 KB data, 4-way, 2-cycle hit latency
- Memory—dual ported with 100-cycle latency

Mis-prediction Validation Unit (MVU):

- IBP buffer—16 KB, 2K entry of IBP (PC, its target address) with 1-cycle hit latency
- IBP buffer indexing—XORing of PC and its target address
- IBP buffer miss penalty—200 cycles or 500 cycles (for IBP table walk)

We considered several IBP buffer sizes, from 512 entries to 8K entries. We charged 200 cycles for the MVU's access of the IBP table on an IBP buffer miss. An IBP table walk on an IBP buffer miss is simpler than the page table walk by MMU on a translation lookaside buffer (TLB) miss for virtual address translation, which is known to take three to five memory accesses because of the hierarchical structure of the page table.

Note that with a 32-bit address space, the page table for a single process has a million entries with a 4-KB page size, whereas the IBP table size is usually on the order of thousands. However, for the sake of generality, we also did our experiments with a 500-cycle penalty for the IBP table access. Recall that MVU accessing the IBP table in memory does not necessarily cause a performance penalty. The program execution will be blocked only if there is no matching IBP, namely, that an attack is detected. However, to have a conservative performance estimate, we have the processor wait in our simulation until the MVU's access of the IBP table in memory finishes, issuing the "validated" signal (as shown in Figure 2).

We observed that with buffer sizes of 1K entries and larger, the IBP buffer provides over 99 percent hit rates for all the benchmark programs. With an IBP buffer of 8K entries, the miss rate is 0.075 percent on average, ranging from 0.001 percent for *eon* to 0.16 percent for *gcc*. Our simulations suggest that the 2K-entry IBP buffer would be the most cost-effective with an average miss rate of 0.097 percent, ranging from 0.001 percent for *eon* to 0.297 percent for *gcc*. We selected the 2K-entry IBP buffer for our performance evaluation experiments.

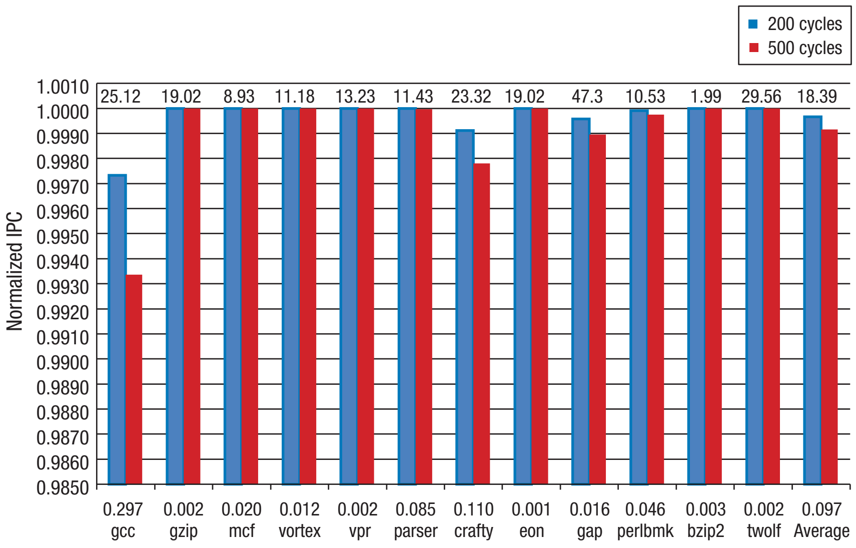

FIGURE 5. MVU performance overhead in terms of normalized instructions per cycle (IPC) over the case without an MVU's control validation. Above the bars for each program is the mis-prediction rate (%) for indirect branches including return, and below the bars for each program is the IBP buffer-miss rate (%).

Three factors determine the MVU performance overhead. The first is the ratio of indirect branch instructions to all instructions. The second is the mis-prediction rate for indirect branch instructions. The third is the IBP buffer miss rate. With an IBP buffer miss, MVU needs to access the IBP table in memory and replace the corresponding IBP buffer entry. Recall that an access of the IBP buffer happens only on a branch mis-prediction for indirect branches. With relatively fewer occasions of encountering indirect branches along with a decent branch prediction success rate, the low IBP buffer miss rates suggest little performance impact by the MVU.

Even with our conservative assumption of stalling the instruction pipeline at every IBP buffer miss, the MVU is shown to be an extremely efficient mechanism (see Figure 5). The performance overhead in terms of normalized instructions per cycle (IPC) is, with a 200-cycle IBP buffer-miss penalty, 0.034 percent on average with the highest overhead of 0.26 for *gcc*. This is an order of magnitude or two less performance overhead than the CFI implementations with added instrumentation for control-flow validation. For example, an early CFI implementation with a static CFG[6] reported the average overhead of 21 percent, while recent coarse grain CFI implementations have reported the average overhead of 1 percent[10] and 2.14 percent.[11] The low performance overhead of the recent CFI implementations comes from using heuristics on indirect branch behavior along with hardware support. However, such heuristics expose vulnerabilities for CRAs.[7] The main contributor to the efficiency of the MVU is pruning the number of indirect branch instruction instances to validate. On average, about 82 percent of the control-flow validation in our experiments was done by the BPU without accessing the IBP table.

In our experiments, the last value prediction scheme was used—storing a single last-seen target address for an indirect branch in the BTB. Moreover, the RAS assumed was a relatively small straightforward design with 16 entries. Branch prediction for indirect branches can improve significantly by providing a separate jump target cache or storing multiple targets in the BTB for an indirect branch.[12] With the improved prediction success rate, the already very low performance overhead will decrease even further.

Our experimental results are in a single process environment and do not consider context-switching effect. Our experiments with a higher IBP buffer-miss penalty of 500 cycles are somewhat reflective of an environment running multiple processes together. However, even with the conservative assumption of stalling the instruction pipeline at every IBP buffer miss, while also using the higher IBP buffer-miss penalty, the performance overhead remains low: less than 0.1 percent on average.

DETECTING CODE REUSE ATTACKS

A code reuse attack based on ROP or its variants will mount the attack by linking the gadgets. Each gadget is a short consecutive portion of a function residing in memory that ends with an indirect branch instruction. With an MVU in place for control-flow validation, a CRA cannot proceed unless only legitimate control-flow transfers are used. Further, the MVU's control-flow validation integrated into the instruction execution pipeline provides less chance for a CRA to bypass the validation than the added instrumentation for existing CFI implementations. For example, in the Intel x86 instruction set architecture (ISA) with variable instruction length, an attack can "create" indirect branches by fetching the instruction byte from the middle of multi-byte instructions.[5] These unintended branch instructions are not protected by the inline code instrumentation for CFI because it is difficult, if not impossible, to guess where to insert the inline code instrumentation to check them.

Most CFG representations suffer from the lack of fully accurate and complete context information.[9] For example, if two returns for two different legitimate call sites for a shared procedure are mixed up in a sequence different from the original program, it will be deemed legitimate because the two IBPs for the two returns are legitimate per the CFG. A CRA that uses only legitimate control-flow transfers per the CFG, but uses a different control-flow transfer sequence from the uncompromised original program, can evade the protection because each control-transfer is considered independent of the other control-transfers.

CRAs have evolved into more sophisticated types, and will continue to do so. With more sophisticated CRAs, more complex fine-grained CFGs are needed. Although incorporating a more complex fine-grained CFG with the MVU needs further investigation, we can note that the context information—such as the conditional branch history or execution path information, in addition to the IBPs—is readily available in BPUs. Using branch prediction in an MVU generally benefits CFI implementations by reducing the need for control-transfer validation, and the benefit will be greater with more complex and context-sensitive CFGs.

In this article, we have considered validating each control-transfer instance as an integral part of indirect branch instruction execution. An MVU working in tandem with the BPU found in most processors validates every control-flow transfer instance in terms of IBP, that is, its indirect branch location and its target address. Since recently encountered and validated IBPs are readily available in the BPU, BPUs do most of the validations, about 82 percent in our experiment, whereas the MVU does the validation for the mis-predicted indirect branches. To enable an MVU to validate the control-flow without delaying the instruction execution pipeline, we have introduced a small buffer (IBP buffer) that holds the IBPs for the indirect branches mis-predicted by the BPU.

Our performance simulations of the SPEC CPU integer benchmark programs via the SimpleScalar simulator show that the MVU incurs a 0.034 percent performance overhead (in terms of the IPC) on average. Moreover, we can disregard this negligible MVU performance overhead because it comes from our conservative assumption that every IBP buffer miss adds a delay to the program execution. However, an MVU's handling of IBP buffer misses does not need to interfere with the program execution in normal situations with no attack—namely, when the control-flow is not compromised. Further, an MVU's integration of control-flow validation into the instruction pipeline is software-transparent, making it more difficult for the attacks to bypass. C

ACKNOWLEDGMENTS

This work was supported in part by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science, and Technology (2015R1A2A2A01).

ABOUT THE AUTHORS

YONGSUK LEE is currently a PhD student in the Department of Computer Science & Engineering at Korea University. His research interests include computer architecture, trusted computing, and system security. Lee received his MS in computer science and engineering from Korea University, Seoul. Contact him at duchi@korea.ac.kr.

GYUNGHO LEE is a professor in the Department of Computer Science & Engineering at Korea University. His research and teaching interests include computer architecture, specifically in the areas of microprocessor architecture, system security, and code optimization. Lee received his PhD in computer science from the University of Illinois at Urbana-Champaign in 1986. He is a fellow of the American Association for the Advancement of Science (AAAS). Contact him at ghlee@korea.ac.kr.

REFERENCES

1. Y.-J. Ahn et al., "Monitoring Translation Lookahead Buffers to Detect Code Injection Attacks," *Computer*, vol. 47, no. 7, pp. 66–72.
2. E. Buchanan et al., "When Good Instructions Go Bad: Generalizing Return-Oriented Programming to RISC," *Proc. 15th ACM Conf. Computer and Communications Security* (CCS 08), Oct. 2008, pp. 27–38.
3. S. Checkoway et al., "Return-Oriented Programming without Returns," *Proc. 17th ACM Conf. Computer and Communications Security* (CCS 10), Oct. 2010, pp. 559–572.
4. F. Schuster et al., "Counterfeit Object-Oriented Programming: On the Difficulty of Preventing Code Reuse Attacks in C++ Applications," *Proc. IEEE Symp. Security and Privacy* (S&P 15), 2015, pp. 745–762.
5. H. Shacham, "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)," *Proc. 14th ACM Conf. Computer and Communications Security* (CCS 07), Oct. 2007, pp. 552–561.
6. M. Abadi et al., "Control Flow Integrity Principles, Implementations, and Applications," *ACM Transactions on Information and System Security* (*TISSEC*), vol. 13, no. 1 (article no. 4), Oct. 2009.
7. E. Goktas et al., "Out of Control: Overcoming Control-Flow Integrity," *Proc. IEEE Symp. Security and Privacy* (S&P 14), 2014, pp. 575–589.
8. J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 2012, Morgan Kaufmann.
9. N. Carlini et al., "Control-Flow Bending: On the Effectiveness of Control-Flow Integrity," *Proc. 24th USENIX Conf. on Security Symp.*, 2015, pp. 161–176.
10. V. Pappas, M. Polychronakis, and A.D. Keromytis, "Transparent ROP Exploit Mitigation Using Indirect Branch Tracing," Proc. 22nd USENIX Conf. Security Symp., 2013, pp. 447–462.
11. M. Kayaalp et al., "Branch Regulation: Low-Overhead Protection from Code Reuse Attacks," *Proc. Int'l Symp. Computer Architecture* (ISCA 12), 2012, pp. 94–105.
12. H. Kim et al., "Virtual Program Counter (VPC) Prediction: Very Low Cost Indirect Branch Prediction Using Conditional Branch Prediction Hardware," *IEEE Transactions on Computers*, Vol. 58, No. 9, 2009, pp. 1153–1170.