Leveraging Speculative Architectures for Run-time Program Validation

Juan Carlos Martinez Santos and Yunsi Fei
Dept. of Electrical and Computer Engineering
University of Connecticut
{juan.martinez_santos, yfei}@engr.uconn.edu

Abstract— **Program execution can be tampered by malicious attackers through exploiting software vulnerabilities. Changing the program behavior by compromising control data and decision data has become the most serious threat to computer systems security. Although several hardware approaches have been presented to validate program execution, they mostly suffer great hardware area or poor ambiguity handling. In this paper, we propose a new hardware-based approach by leveraging the existing speculative architectures for run-time program validation. The on-chip branch target buffer (BTB) is utilized as a cache of the legitimate control flow transfers stored in a secure memory region. In addition, the BTB is extended to store the correct program path information. At each indirect branch site, the BTB is used to validate the decision history of conditional branches before it, and more information about the future decision path is fetched to monitor the execution path at run-time. Implementation of this approach is transparent to the upper operating system and programs. Thus, it is applicable to legacy code. Due to good code locality of the executable programs and effectiveness of branch prediction, the frequency of run-time control flow validations against the secure off-chip memory is low. Our experimental results show a negligible performance penalty and small storage overhead with ambiguity reduced.**

I. Introduction

As networking connections become pervasive for computer systems and embedded software contents increase dramatically, it has been more convenient for hostile parties to exploit software vulnerabilities to attack computer systems. Typical security attacks include buffer overflows, fault injections, Trojan horses, and data and software integrity attacks. The buffer overflow vulnerability allows user data to overwrite program code or control-flow structure [1]. Hence, the program execution can be directed to the malicious code that is implanted into the program's memory space, opening part or all of the system to the adversary. Some worms, such as the Morris Worm, the W32/Blaster, and the Code Red worm, have taken advantage of the buffer overflow vulnerability and resulted in serious virus intrusion or even denial of service. Reports from CERT [2] show that control data attacks are the most dominant and critical security threats today.

A control data attack compromises control flow transfers, e.g., at procedure call, return, local jump/branch, special non-local jump (setjump/longjump). Since it only alters the control data (target address) but still follows the instructions' semantics without explicitly violating any security policies, traditional measures such as code/data integrity checking or encryption/decryption alone can not prevent the control data attacks. Recently there are some software-based mechanisms presented to countermeasure control data attacks, such as StackGuard, StackGhost, and RAD [3], [4], [5], [6], [7]. They suffer a significant performance penalty and normally require use of standard dynamic libraries and the source code to be available. Other hardware-based schemes are either based on observations of violation symptoms [8], [9], [10], [11] or tracking control source data [12], [13]. Their effectiveness, however, is often limited by inaccurate information being monitored or large memory/performance overhead.

Recently, some exploits have emerged that can change the direction of control instructions instead of the target address by overwriting local variables[14], [11], [15], which is called *decision data attack*. Although all the control transfers are valid, the global control flow is compromised, and the attackers can extract important system information. This kind of attacks is hard to detect by control flow transfer validation only. Therefore, we have to validate the branch decision path at run-time as well.

In this paper, we introduce a novel approach of protecting program execution at the micro-architecture level. We propose to monitor control flow transfers and execution paths at the super block level [1]. At each checking point (indirect branch site), dynamic program execution information is compared against a full record set (FRS) stored in a secure memory region, including legitimate target addresses and execution paths. In addition, behavior reference for future program execution is also prefetched for later monitoring. To enable fast validation, we consider introducing a cache architecture for the FRS in memory. As branch prediction scheme has been widely adopted in embedded processors, the on-chip branch target buffer (BTB) can be used as a perfect cache for the FRS. We propose a mechanism to ensure security for the BTB, i.e., avoiding any possible pollution from tampered memory. We find that the validation mechanism only needs to be activated for those indirect branches that are mis-predicted by the BTB, in terms of direction, address, or execution

[1]A super block is a portion of code between two consecutive indirect branch instructions.

history. The rare access of the external FRS results in very little performance degradation than other hardware schemes. The modification to the processor architecture is minor on top of the branch prediction structure and is transparent to the upper operating system and programs. Thus, legacy code can run on the secure processor without recompilations.

The remainder of the paper is organized as follows. We first describe the motivation of our control flow transfer and execution path validation in Section II. Section III presents details of the proposed mechanism and architecture for run-time validation based on the BTB architecture. Section IV evaluates the performance impact of our approach. Section V compares related work to our contributions. Finally, Section VI draws conclusions.

II. Motivation

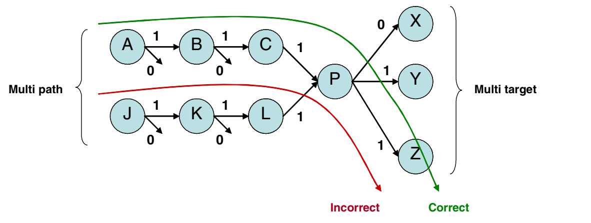
When control flow attacks are inflicted on a system, the program execution can be directed to either malicious code injected by the attacker or some existing code that would not otherwise execute at this moment. It is the execution of the compromised control instructions that deviate program execution from its expected behavior. To prevent control flow attacks, our micro-architecture level mechanism validates both control flow transfers and execution paths. The program execution monitoring is sampled at run-time at the sites of indirect branches, instead of every conditional branch or jump [11]. Thus, the performance and storage overhead incurred by the monitoring may be reduced greatly. However, since the checking is carried out less frequently, the coverage of each checking process has to be adequate for the program to reduce the false negative rate of validation.

A. Complex Program Control Flow

There are several issues in program execution that complicate the validation process of control flow transfers and execution paths, e.g., multiple-path situations, as shown in an example control flow graph in Fig. 1, where each node represents a basic block, and each edge is a control flow transfer. Here basic block P ends with an indirect instruction that may go to different target addresses for taken branches. There are multiple paths of basic blocks that lead to block P. Without proper handling, they may result in ambiguities that will increase the false negative rate and degrade the effectiveness of the security countermeasures.

For direct conditional branches, a program control flow can be validated by using only binary decision information (where 1 represents branch taken and 0 branch untaken). However, when indirect branches are present, validation has to be performed against the address path, which requires a lot of memory storage [11]. An alternative solution is to look backward at the history of branch decisions at indirect branches, and a binary decision path is used [15]. However, there are ambiguities arising in this approach. In Fig. 1, there are two paths that lead to basic block P which is the end of a super block; assume the path of $A-B-C-P$ is correct, and the other path of $J-K-L-P$ is infeasible. However, their binary history paths are the same: 1-1-1. If we just use the binary history path information, 1-1-1 will be stored in the signature and the incorrect path of $J-K-L-P$ will escape checking. Therefore, the decision history path is insufficient to validate the execution, although it uses much less storage than the address path.

We propose a hybrid solution for the above situation: each history path has to be associated with its past branch site (PBPC). As shown in the last column of the table in Fig. 1, the correct path for basic block P associated with TPC of Z will be C-1-1-1, a hybrid of direction history path and the last branch site address. We have modified the SimpleScalar toolset [16] to profile a set of MiBench applications [17], to find how common the ambiguities for binary paths are. For example, for application *patricia*, 5 indirect branch sites are found to have ambiguous paths that share the same binary history, and each site has 2 ambiguous paths. In run-time execution, these ambiguous paths can be executed thousands of times due to loop iterations, function calls, etc. However, after considering the PBPC together with the binary history path, the ambiguity is fully dissolved for *patricia*.



	Address Path [32 x n bits]	Branch History [n bits]	Our Approach [(32+n) bits]
Correct Path	A-B-C-P	1-1-1	C-1-1-1
Incorrect Path	J-K-L-P	1-1-1	L-1-1-1

Fig. 1. Complex program control flow with ambiguity

In addition, the information on future expected paths (EP) is also obtained at the end of each super block, which will be checking the decision of basic blocks in the next super block. Instead of storing all the possible binary paths, we can get an all-path-vector for the current TPC for a number of levels or until the program execution reaches the end of the super block. Fig. 2 shows one example of an expected path vector for a depth of 2. Basic block A is the end of last super block. One of the taken branch of A goes to B, and B will be the root of a binary tree, because all the basic blocks between B and next super block are direct conditional branches, and their directions are sufficient for path validation. With a depth of 2, there are 4 paths in total: $B-C-E$ (with a decision path of 11), $B-C-F$ (10), $B-D-G$ (01), and $B-D-H$ (00), and we use a 4-bit vector to represent the validity of each path. For example, if path $B-D-G$ (decision history path is 01) is invalid and all the other three paths are valid, the vector will be 1101. For depth n, the vector size is 2^n bits, where each bit represents the validity of its corresponding path, and the vector size is much smaller than the maximum size of $2^n \times n$ bits for recording all the possible n-level paths.

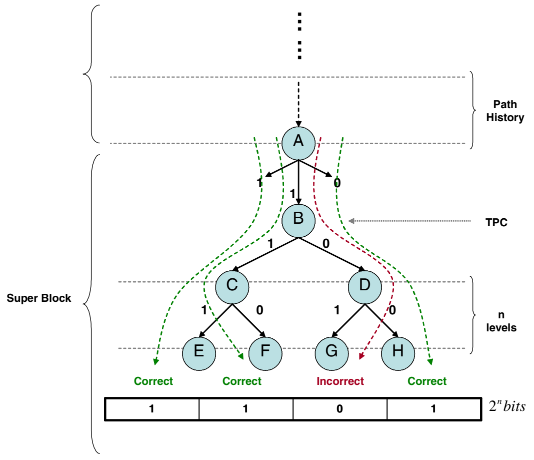


Fig. 2. Expected paths vector fetched at an indirect control instruction site

The vector is used during the program execution to validate it on the fly, making faster detection of attacks possible. Whenever a conditional branch is encountered and resolved, its direction decision (taken or untaken, i.e., 1 or 0) is used to cut the vector in half, with the higher half kept for decision 1 or the lower half for 0. If the vector reduces to 0, an invalid path has been executed and the anomaly is detected. For example, in Fig. 2, the 2-level all-path-vector of node B is 1101. Assume the program is executing path $B-D-G$. When the control flow transfer of $B-D$ is resolved, the 0 decision keeps the lower half of the vector, 01. When the program proceeds to the next level, $D-G$ transfer is taken (decision is 1), and the higher half of 01, 0, is kept. As 0 is detected, the execution of the invalid path $B-D-G$ is caught by the monitoring mechanism.

B. Training the Full Record Set (FRS)

The full record set (FRS) in a secure memory region represents the normal behavior of a program, which theoretically contains all the legitimate indirect control transfers and execution paths. There are two ways to collect the records[18]. We can either extract the normal behavior through static analysis of the legacy code, or perform training as many models-based approaches have done [11]. By running the applications in certain secure environment with a lot of test data, the processor can regard all seen control flow transfers and execution paths as normal ones. If the full record does not completely cover the normal behavior, using it as a reference to validate program execution at run-time will incur false alarms (false positives). Previous works have shown that the number of control flow transfers actually converges quickly as the number of indirect control instructions increases [18]. In our study, we use train-data inputs for SPECINT benchmarks to profile them and to get their FRS.

As our approach is sampling the program execution checking only at indirect control instruction sites, it is important to make each checking cover a broad range of program execution to improve the detection rate of invalid control flow transfers and anomalous execution paths. For each super block, the execution path is first validated against the expected path vector; then at the end of the super block, the dynamic branch decision history is validated against the history paths pre-characterized. Both the expected path and the history path should be long enough to cover most super blocks. However, the size of a super block varies greatly in programs. We examine the average size to determine the appropriate path length. Based on our profiling study, Fig. 3 shows the ratio of the number of conditional branch executions over the number of indirect control instruction executions for several SPECINT benchmarks. The ratio varies greatly with applications, but for most benchmarks, there is one indirect instruction executed for every 5-6 conditional branches. We have set the length of history paths to be 14 conditional branch, and the depth of expected path vector to be 6, so that the total length is 20.

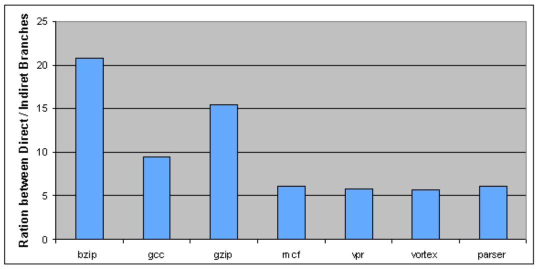


Fig. 3. Ratio of the number of conditional branch executions over number of indirect instruction executions for SPECINT

III. Speculative Architectures for Control Flow Transfers and Execution Paths Validation

For every indirect control instruction, the system has to validate its legitimacy. Since the full record set (FRS) is stored in a secure memory region, frequent off-chip full record access for validation will degrade program performance greatly. To reduce the performance overhead, we can either reduce the latency for accessing the full record or decrease the number of accesses. Previous work has employed a Bloom Filter scheme to reduce the off-chip hardware access latency to 4 cycles [18]. In this paper, we focus on reducing the access frequency by leveraging the speculative architecture of branch target buffer (BTB).

A. Extraction of Dynamic Program Execution Information for Validation

When validating program execution, the system needs to collect dynamic information on the history of conditional branch directions and the last branch address to compare against the normal behavior obtained from static time analysis or training. We dedicate two registers for dynamic information, which can only be accessed and used by the validation system. The branch history shift register (BHSR) [15] is a shift register that stores the decisions of past n conditional instructions (assume the length of BHSR is *n*), and it is updated after each direction is resolved. Another register, the past branch program counter (PBPC), is to record the branch address of the last basic block.

At each indirect instruction site, the branch site (BPC) is used to look up the expected behavior either in the BTB or in the FRS. If a BTB entry or an *Indirect Control Signature* (ICS) [2] in the FRS is found that matches the tuple of $\{$BPC, TPC, BHSR, PBPC$\}$, the target address and history are valid. Otherwise, there is a mismatch, and an alarm is raised for the operating system to take control over the program. In the mean time, the expected path vector is fetched to check the following basic blocks along the execution, and alarm may be raised any time within the next super block if the vector is reduced to 0.

B. BTB Update and Administration

Branch prediction mechanisms have been widely adopted for high-end embedded processors to alleviate pipeline stalls caused by conditional instructions. Predicting the branch direction while caching the branch target address has been the fundamental objective in many designs of efficient pipelined architectures. The branch target address is provided by the BTB, which has a cache structure consisting of 512 sets with the set associativity range from 1 to 8 ways [19]. The upper part of the BPC is used as a tag to access the BTB.

Fig. 4 illustrates the branch prediction flow using a BTB in a simple five-stage pipeline architecture. The solid objects and lines are for the regular branch prediction scheme. The flow is also extended with some enhancements for control flow transfer and execution path validation, as shown in dashed lines and objects, which will replace the original operations.

When an instruction is fetched from the instruction memory (in *IF* stage), the same PC address is used to access the BTB. A hit at this point indicates that the instruction to be executed is a control instruction. The predicted target PC (TPC) is then used as the next PC (NPC) to fetch a new instruction in the next pipeline stage (*ID*), rather than waiting until the later stage (*EX*) to use the computed NPC for fetching, to avoid branch stalls. Meanwhile, according to the instruction type, a direction prediction or computation is performed. If the branch is computed to be taken (*direction hit*), the TPC from the BTB will be compared with the computed NPC in the *EX* stage. If these two values match (*address hit*), both the branch direction and the target prediction are correct. However, one more validation has to be done for indirect control instructions in our approach.

The system has to compare the BHSR and PBPC against the HPs in the BTB entry. If the history matches as well, it is a *history hit*, and the program execution will continue as normal. Otherwise, it is a *history mis-prediction*: the history paths associated with the TPC in the BTB do not include the history just seen. The tuple of $\{$BPC, computed NPC, BHSR, PBPC$\}$ has to be sent to the external memory for further validation. Only when it misses again, an security alarm is raised.

In traditional architecture, on an *address mis-prediction* site, the BTB entry is just updated when the next PC is resolved from the instruction. However, since an indirect target address mis-prediction may also be caused by security attacks, in our enhanced architecture, the external memory has to be checked before the corresponding entry in the BTB is updated with the matched ICS. At a *direction mis-prediction* site where there is a BTB entry for the instruction but the instruction is actually not taken, the entry is deleted and the fetched TPC is squashed. There is normally a mis-prediction penalty for these remedy actions.

On the leftmost side of the flow diagram, if no entry is found in the BTB for the current PC, the instruction may be a regular data path instruction or a control instruction falling through. In the subsequent ID stage, if the instruction is found to be a taken indirect control instruction, it is a *direction mis-prediction* and the address is not cached in the BTB. Before a new entry is inserted to the BTB, the tuple of $\{$BPC, computed NPC, BHSR, PBPC$\}$ has to be validated against the record in memory. Since the BTB has limited capacity, a replacement policy, e.g. least recently used (LRU), may be employed to find a victim to evict for multi-associativity BTB.

A lot of studies have been presented on improving prediction accuracy to reduce the mis-prediction penalty and thus performance degradation. We observe that for regular speculative architecture, the BTB has served as a target address cache for some of the taken branch instructions, and mis-prediction just affects performance. To turn the BTB into a cache for the full record, we have to first extend it to include the path information as well. More importantly, we have to ensure its integrity, i.e. guarantee the BTB is free of corruption from the external memory. Thus, when we use the BTB as the reference to validate control flow transfers and execution paths at run-time, on any mis-prediction site, including direction, target address, and history path, we have to look up the external full record for the current BPC before the BTB is updated.

C. Architecture Support for Control Flow Validation

Access to the full record should be reduced to the minimum to lower the performance overhead. Thus, a clear distinction has to be made between instructions that will point to a safe target address and the instructions that definitely require validation against the external FRS. Assuming that code integrity checking mechanisms, e.g., [20], [21], have been applied, and based on the analysis in Section II, we can regard direct control instructions always generating

[2]Each Indirect Control Signature (ICS) consists of one branch site (BPC) as a tag, one target site (TPC), multiples history paths (HPs), and the vector for the expected paths (EPV).

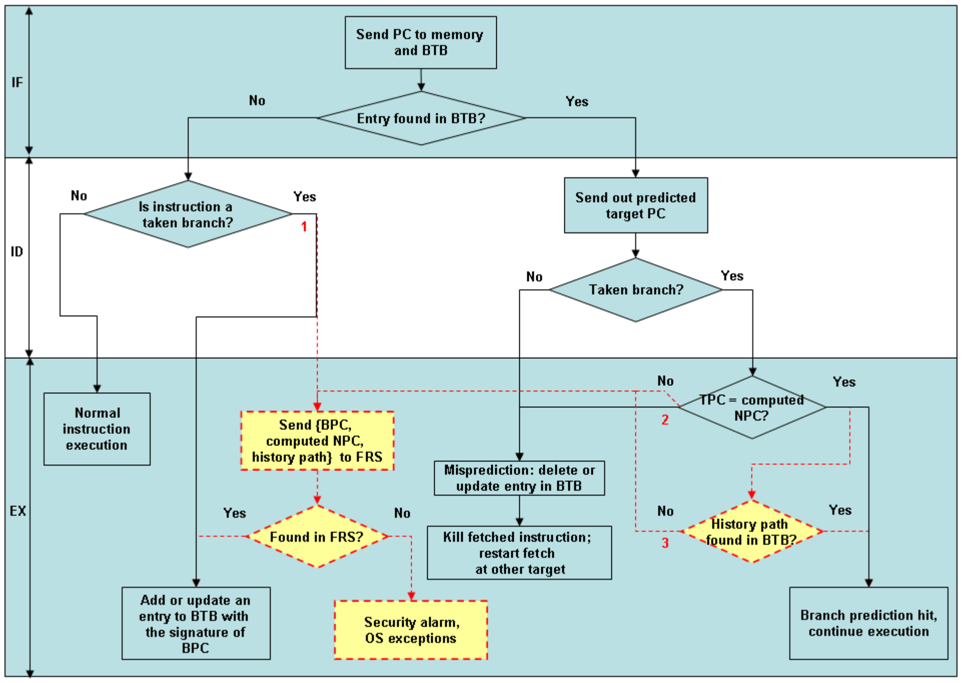

IF
ID
EX
Send PC to memory and BTB
Entry found in BTB?
No
Yes
Is instruction a taken branch?
Send out predicted target PC
Taken branch?
1
2
3
Normal instruction execution
Send {BPC, computed NPC, history path} to FRS
TPC = computed NPC?
Misprediction: delete or update entry in BTB
Found in FRS?
Kill fetched instruction; restart fetch at other target
History path found in BTB?
Add or update an entry to BTB with the signature of BPC
Security alarm, OS exceptions
Branch prediction hit, continue execution

Fig. 4. Regular branch prediction flow enhanced with security features

correct target addresses because the addresses are encoded in the instruction and the instruction bits have been ensured correct. Hence, on BTB mis-predictions, these instructions can directly update the BTB with the computed next target address without validating against the full record set. Note that only target address is needed for their BTB entries. Also, the full record in memory does not need to keep any information for direct control instructions. Only those indirect control instructions that are mis-predicted by the BTB will possibly incur full record lookups.

Fig. 4 shows that at three sites (labeled 1, 2, 3), which represent direction mis-prediction, address mis-prediction, and history mis-prediction respectively, the tuple of {BPC, computed NPC, BHSR, PBPC} is sent to the full record for validation. Fig. 5 illustrates the overview of our architecture support in processor pipelines for control flow validation. For every indirect branch instruction, the BPC is sent to the BTB during the fetch stage. The dynamic information extracted, including next target PC, BHSR, and PBPC, is sent to the BTB for validation after the execution stage. On any mis-prediction, the full record set (FRS) in the secure memory is accessed, and the corresponding indirect control signature (ICS) is brought into the processor. With the expected path vector fetched into the processor pipeline, the program execution is monitored at run-time on a basic block basis.

Our BTB entry has a fixed size for the sake of simplicity. Consequently, the lengths of the history paths and expected path vector in the BTB entry are also pre-defined. However,

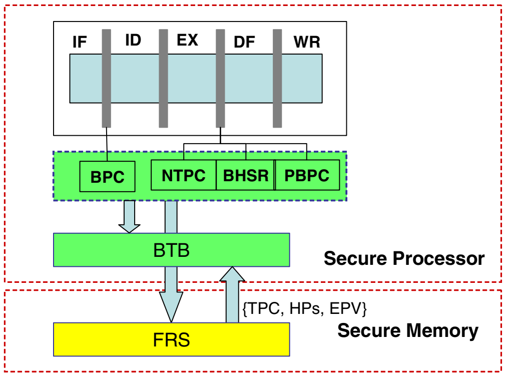


Fig. 5. The architecture support in processor pipeline for control flow validation

due to the varying size of super blocks, the actual length we can extract dynamically is not fixed. We use a counter to record the number of basic blocks in a super block, and use it to mask the comparison between the dynamic BHSR and the static HPs in BTB. The number of bits in each BTB entry for an expected path vector is 2^n if the preset number of levels is n. Assume the depth of a super block is m, and $m < n$, we only get an all-path-vector of 2^m bits. The vector is extended by duplicating each bit $2^{(n-m)}$ times, in order to be put in a BTB entry for run-time validation. Fig. 6 shows an example where n=4 and m=2, thus, each bit in the vector obtained from training is duplicated $2^{(n-m)} = 2^2 = 4$ times to compose the BTB EPV.

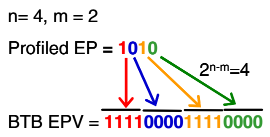


Fig. 6. Extending the all-path-vector from profiling for the BTB entry

IV. Experimental Results

We test a set of SPECINT applications running on a modified cycle-accurate SimpleScalar [16] that models an in-order single-issue processor to measure the performance impact of our approach. The architecture parameters for the simulator are listed in Table I. We consider extra delays for accessing the full record set for validation when the branch prediction unit is enhanced with security features.

TABLE I
Architecture Parameters for Simulations

Parameter	Value
BTB	512 sets, n-way set associative
Return address stack	8 entries
Branch miss penalty	3 cycles
Branch predictor	Bimod
Full record set access latency	50/100/150/200 cycles
Fetch/dispatch/issue width	1
Pipeline stages	5
Load/store queue	8 entries
Memory access latency	18 cycles

A. Storage Overhead

Compared to the conventional BTB, our enhanced BTB should contain reference path information as well. For the length that we have chosen, i.e., 6 for the expected path vector and 14 for the history path, each BTB entry will be 174 bits long, 110 bits more than the traditional structure.

Also, as a FRS is needed in a secure memory region, the memory footprint for the programs will be increased as well. Fig. 7 shows the results for a set of SPECINT benchmarks, where for each application, the first bar represents the original program code size and the second bar represents memory overhead. We can see that the memory overhead for most of the benchmarks is quite small, a few kilo bytes, with the exception of *gcc* which requires 250KB. This is due to the large number of tuples of {BPC, TPC, HPs}. The average ratio of memory overhead over the program code size is 3.79%.

B. Performance Impact of the Run-Time Validation Mechanism

We next examine the performance impact of our run-time validation mechanism. Our BTB configuration contains three parameters, i.e., the number of sets, the set associativity, and the number of history paths in each entry (the path

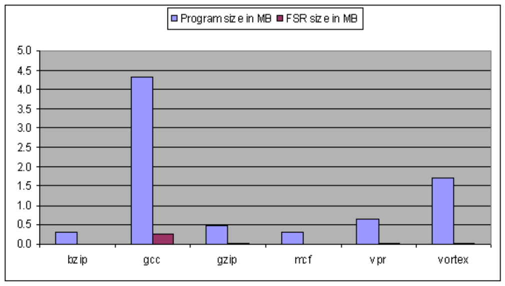


Fig. 7. Memory overhead for the Full Record Set (FRS)

associativity). Fig. 8 demonstrates the resulting performance degradation for a set of SPECINT benchmarks with the BTB configuration of 512 sets, direct map, and 1 history path in each BTB entry. Looking up the full record set in memory to validate mis-predicted indirect instruction targets has resulted in negligible performance overhead. Even with the full record memory access time set at 200 cycles, for most of the benchmarks, the execution cycles overhead is 4.82% in average. For application *gcc*, the overhead is noticeable - up to 24.13%.

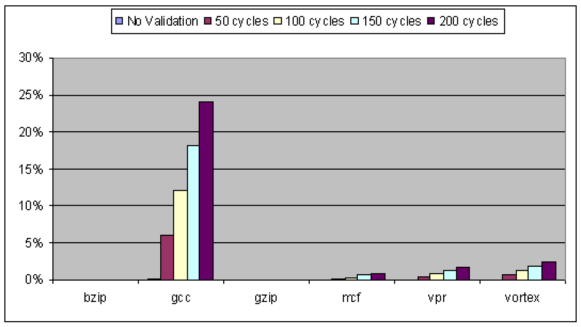


Fig. 8. Performance degradation of our validation mechanism for the BTB configuration of 512-1-1

We next check the impact of each BTB parameter on the performance. We change the number of sets but keeping both the set associative and history path associative as 1. Fig. 9 shows the simulation results for BTBs with the number of entries as 512, 1024, 2048, and 4096. The performance overhead cycles are normalized to the case for a BTB with 512 sets. For benchmarks *gzip* and *mcf*, the overhead cycles actually increase as the capacity of BTB increases. The result is very counter-intuitive. Because for the conventional BTB architecture, the performance should improve as the size increases [19]. However, since our enhanced BTB is used not only for control flow transfer validation (TPC), but also for path validation, there may be more memory access due to BTB cold miss as the BTB size increases. For other benchmarks, there is no big difference for the performance as the BTB size increases. The reason is that the total number of indirect instructions in the program is less than 512, so that the performance overhead already saturates. For *mcf*, the overhead starts saturating at 1024, which indicates that the number of indirect instructions for it is less than 1024.

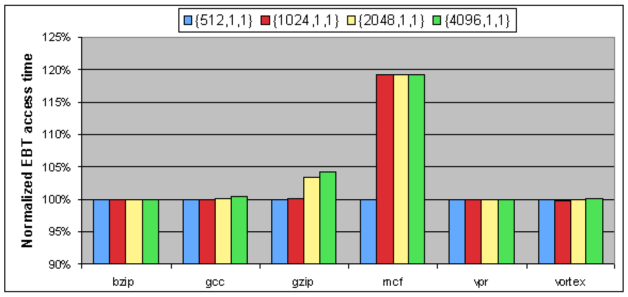


Fig. 9. Normalized overhead with different BTB size to the case for a BTB of 512-1-1

We then change the BTB set associativity but keep the other two parameters constant. Fig. 10 demonstrates the resulting execution cycle overhead with different set associative (n = 2, 4, 8) normalized to that of the direct map case. The result is similar to the previous one. As the set associativity increases, the effective capacity of the BTB increases, and the overhead also increases or saturates due to the similar reasons.

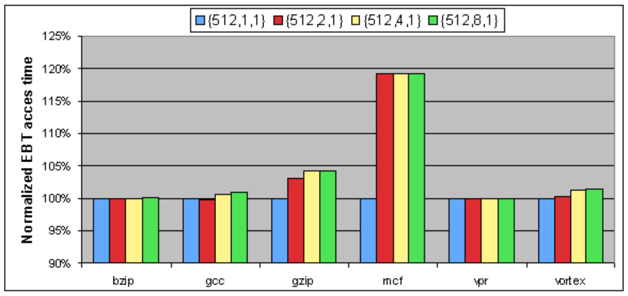


Fig. 10. Normalized overhead with different set associativity to the case for a BTB of 512-1-1

We finally change the BTB history path associativity but keep the other two parameters constant. Fig. 11 demonstrates the resulting execution cycle overhead with different number of history paths in an entry (n = 2, 4, 8) normalized to the case with 1 history path per BTB entry. We see that as the history associativity increases, the performance overhead drops dramatically, especially for benchmark *bzip*. Note that since the number of BTB sets and associativity do not change, the BTB capacity is the same, and there is no increase in BTB cold misses. However, with more history paths associated with a selected TPC, the chance of finding a matched HP in the BTB is increased, and thus the external memory access time is reduced. When the overhead time becomes saturated at some point, for example, n=4 for *gcc*, it means that most of the indirect branches have less than 4 history paths leading to it. For *mcf* and *bzip*, n=2 is enough.

V. Related Work

There has been a lot of research work on using specialized software and hardware to detect specific attacks such as buffer overflows [13], [5], [7], [10]. These techniques focus

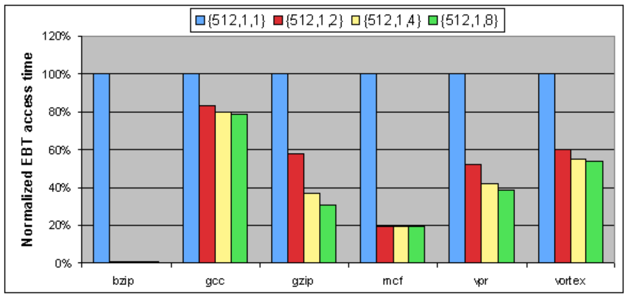


Fig. 11. Normalized overhead with different history associativity to the case of a BTB of 512-1-1

on the attack itself and can prevent attackers from exploiting the particular vulnerability. They fall in the category of white-box approaches, where the attack mechanism has been analyzed in detail and particular precautions are taken. However, due to many sources of vulnerabilities, it is desirable to employ a more general symptom-based mechanism to detect any abnormal operations. This is a kind of black-box approach, where the program execution is monitored at run-time and any deviation from its legitimate behavior is detected, no matter what is the cause.

Researchers have shown that the program behavior can be modeled at different granularity levels, including system call, function call, control and data flow. Simply checking system calls is not sufficient and may cause high false negatives in some cases. Also, storing them in a Finite State Automata (FSA) and checking against all normal system call traces is a tremendous design effort[22], [23], [24]. Some previous works have focused on increasing the reliability of return address stack (RAS) to defeat the exploits at the function call level [25], [8], [9]. However, the control data for other non-RAS indirect branches can also possibly be tampered by adversaries through contamination of memory. Other finer granularity levels have been proposed to detect abnormal program behavior. There exists some research work that detect control flow errors either by software or hardware support. The software-based approach rewrites the binary code with a checking mechanism in every basic block [3]. Although this technique is flexible, it requires binary translation and increases both the program size and execution cycles dramatically. Hardware support for control flow transfer validation is more efficient in performance, e.g., the hardware-assisted preemptive control flow checking [26]. However, the hardware cost is large because they have to duplicate the whole register file and the PC for validation. Also, illegal indirect branch transfers may slip the checking mechanism. Another hardware-based approach mainly focuses on direct jumps and uses a sophisticated co-processor for the complex control flow modeling and checking [11]. The storage overhead for reference behavior is also very large.

The most related previous work is the Bloom Filter-based run-time control flow validation [18]. They focus on reducing the storage size for legitimate control flow transfers and thus the hardware access latency. However, their Bloom Filter may introduce false negatives. In their follow-up work [15], validations are only performed at indirect branch sites with binary history path (branch direction). This approach may result in a high false negative rate because it cannot solve path ambiguities due to the binary representation. Our approach reduce the ambiguity by associating the binary history path with the last branch address. In addition, our mechanism considers branch correlation between adjacent super blocks by correlating the history path with the future expected path. Thus, our approach should have higher anomaly detection rate than previous work in [15]. By utilizing the existing branch target address prediction mechanism, our mechanism achieves negligible performance overhead even with a long latency for accessing the FRS.

VI. Conclusion

In current processors, control flow transfer is a blindfold without any validity check. With more software vulnerabilities being exploited by adversaries, control data and decision data attacks have become the most dominant and serious threats to computer system security. In this paper, we have proposed a practical architecture-based approach for run-time control flow transfer and execution path validation. With the aid of speculative architecture of branch target buffer (BTB), we narrow down the insecure instructions to indirect control instructions, and sample the validations only at indirect control sites. The anomaly detection rate of our approach should be higher than previous related work because our approach not only validates the history path, but also monitors the next branch decisions at run-time. It results in very little performance overhead with some storage overhead. The future work will include experimenting with fault injections to evaluate the anomaly detection rate of our approach.

References

[1] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, no. 49, Nov. 1996.

[2] "CERT Security Advisories," `http://www.cert.org/advisories`.

[3] E. Borin, C. Wang, Y. Wu, and G. Araujo, "Dynamic binary control-flow errors detection," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 5, pp. 15–20, Dec. 2005.

[4] T.-C. Chiueh and F.-H. Hsu, "RAD: A compile-time solution to buffer overflow attacks," in *Proc. Int Conf. Distributed Computing Systems*, Apr. 2001, pp. 409–417.

[5] C. Cowen, C. Pu, D. Maier, H. Hinton, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks," in *Proc. USENIX Security Symp.*, Jan. 1998, pp. 63–78.

[6] M. Frantzen and M. Shuey, "StackGhost: Hardware facilitated stack protection," in *Proc. USENIX Security Symp.*, Aug. 2001, pp. 55–66.

[7] C. Pyo and G. Lee, "Encoding function pointers and memory arrangement checking against buffer overflow attacks," in *Proc. Int. Conf. Information & Comm. Security*, 2002, pp. 25–36.

[8] R. Lee, D. K. Karig, J. P. McGregor, and Z. Shi, "Enlisting hardware architecture to thwart malicious code injection," in *Proc. Int. Conf. Security in Pervasive Computing*, Mar. 2003, pp. 237–252.

[9] Y. Park, Z. Zhang, and G. Lee, "Microarchitectural protection against stack-based buffer overflow attacks," *IEEE Micro*, vol. 26, no. 4, pp. 62–71, Jul/Aug 2006.

[10] N. Tuck, B. Cadler, and G. Varghese, "Hardware and binary modification support for code pointer protection from buffer overflow," in *Proc. Int. Symp. on Microarchitecture*, Nov. 2004.

[11] T. Zhang, X. Zhuang, S. Pande, and W. Lee, "Anomalous path detection with hardware support," in *Int. Conf. Compilers, Architecture, & Synthesis for Embedded Systems*, Sept. 2005, pp. 43–54.

[12] J. R. Crandall, S. F. Wu, and T. Chong, "Minos: Architectural support for protecting control data," *ACM Tran. Architecture & Code Optimization*, vol. 3, no. 4, pp. 359–389, Dec. 2006.

[13] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *ACM Proc. Int. Conf. Architectural Support for Programming Languages & Operating Systems*, Oct. 2004, pp. 85–96.

[14] H. H. Feng, J. T. Giffin, Y. Huang, S. Jha, W. Lee, and B. P. Miller, "Formalizing sensitivity in static analysis for intrusion detection," in *Proc. IEEE Symp. on Security & Privacy*, 2004.

[15] Y. Shi and G. Lee, "Augmenting branch predictor to secure program execution," in *IEEE/IFIP Int. Conf. on Dependable Systems & Networks*, June 2007.

[16] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *IEEE MICRO*, vol. 35, no. 2, pp. 59–67, Feb. 2002.

[17] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *IEEE Int. WkShp on Workload Characterization*, Dec. 2001, pp. 3–14.

[18] Y. Shi, S. Dempsey, and G. Lee, "Architectural support for run-time validation of control flow transfer," in *Int. Conference Computer Design*, Oct. 2006.

[19] C. Perleberg and A. J. Smith, "Branch target buffer design and optimization," *IEEE Trans. on Computers*, vol. 42, no. 4, pp. 396–412, Apr. 1993.

[20] Y. Fei and Z. J. Shi, "Microarchitectural support for program code integrity monitoring in application-specific instruction set processors," in *Proc. Design Automation & Test Europe Conf.*, Apr. 2007, pp. 815–820.

[21] H. Lin, X. Guan, Y. Fei, and Z. J. Shi, "Compiler-assisted architectural support for program code integrity monitoring in application-specific instruction set processors," in *Proc. Int. Conf. Computer Design*, Oct. 2007.

[22] S. Mao and T. Wolf, "Hardware support for secure processing in embedded systems," in *Proc. Design Automation Conf.*, June 2007, pp. 483–488.

[23] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A sense of self for UNIX processes," in *Proc. IEEE Symp. on Security & Privacy*, 1996.

[24] C. Michael and A. Ghosh, "Using finite automata to mine execution data for intrusion detection: A preliminary report," in *Proc. Int. WkShp on Recent Advances in Intrusion Detection*, 2000, pp. 66–79.

[25] D. Ye and D. Kaeli, "A reliable return address stack: Microarchitectural features to defeat stack smashing," in *Proc. WkShp Architectural Support for Security & Antivirus*, Mar. 2005, pp. 73–88.

[26] R. Ragel and S. Parameswaran, "Hardware assisted preemptive control flow checking for embedded processors to improve reliability," in *Int. Conf. Hardware/Software Codesign & System Synthesis*, Oct. 2006, pp. 100–105.