Microprocessors and Microsystems 33 (2009) 461–468

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Indirect Branch Validation Unit

Gyungho Lee [a,*], Yixin Shi [b,1], Hui Lin [b,1]

[a] *College of Information and Communications, Korea University, Seoul, Republic of Korea*
[b] *ECE Department, University of Illinois at Chicago, 851 South Morgan St. Chicago, IL, USA*

ARTICLE INFO

Article history:
Available online 18 September 2009

Keywords:
Control data
Indirect branch
Branch prediction
Software attack

ABSTRACT

This paper presents a micro-architectural enhancement, named Indirect Branch Validation Unit (IBVU), to prevent malicious attacks from compromising the control data of the program. The IBVU provides a run-time control flow protection by validating a dynamic instance of an indirect branch's address and its target address – *indirect branch pair* (IBP), which represents the program behavior. To validate an IBP at run-time with little performance and storage overhead, the IBVU employs a Bloom filter, a hashing based bit vector representation for querying a set membership. Two organizations trading off of the access delay and space in VLSI design are provided, and three commonly used hashing schemes are evaluated for the performance impact as well as the area overhead. Recognizing potential false positives from adopting the Bloom filter, consideration of reducing it per the Bloom filter's design parameters is discussed, while the difficulty of utilizing the false positives due to hashing based indexing of the Bloom filter for malicious attack is noted.

1. Introduction

There are various ways to exploit software vulnerabilities (such as buffer overflow, format string vulnerability, heap overflow and integer overflow) for malicious attacks. An external adversary always subverts victim program's normal execution by overwriting critical data in the program's address space. By overwriting control data, the data that are used as the target of a control flow transfer instruction, software attacks tend to redirect the control flow to the attacker's way [4]. Meanwhile, there are also attacks overwriting other critical data, such as decision-making data.

Current micro-architecture allows an indirect branch to use any value loaded into the program counter as the destination, lack of checking its validity. It is one of the fundamental reasons that make it difficult for a higher-level solution built upon this unsecured hardware to guarantee every control flow transfer as intended. At machine instruction level, high-level descriptions of program behavior are ultimately translated into control flow transfer instructions of direct branches and indirect branches. We therefore propose a hardware defense mechanism to validate indirect branch's behavior to restrain it from jumping to arbitrary targets. The shifting of the defense into the instruction level not only provides a more secure hardware platform. However, at the machine instruction level protection, the performance and storage overhead needs to be minimal, which makes many existing protection schemes inappropriate.

In order to represent the dynamic execution of the indirect branch, we propose the Indirect Branch Validation Unit (IBVU) to contain the indirect branch behavior signature that includes the target address and the PC address of the interested indirect branch instances. For the IBVU to check an *indirect branch pair* (IBP), i.e. an indirect branch instruction's address and its target address, one needs to represent a set of legitimate IBP's. One may consider a hardware table along with its caching, but it may incur a little excessive performance and storage overhead. As a result, we consider a time-and-space efficient bit vector representation. A Bloom filter, facilitating the indirect branch validation for processor core, is adopted to implement the IBVU. Bloom filter is a space-efficient probabilistic data structure that is used to test whether an element is a member of a set [2]. Thus, the IBVU can efficiently detect at run-time whether a given indirect branch instance is legitimate or not. However, false positives are possible in the Bloom filter due to the indexing of the filter through hashing. As a result, it is possible for the filter to pass some invalid bit patterns as the legitimate ones instead of filtering them out.

Although the false positives are inherent cost of the Bloom filter for the benefit of a fast and efficient representation of program control flows, one should note that the ***false positives from the Bloom filter are NOT directly translated attack vulnerability of escaping the protection***. Note that the false positive rates in the Bloom filter are just a probability that there can be hashed values that hit the same positions of the bit vector in Bloom filter. Encountering the same hashed values from two different control flow

* Corresponding author. Tel.: +82 2 3290 4843.
E-mail addresses: ghlee@korea.ac.kr (G. Lee), yshi7@uic.edu (Y. Shi), hlin33@uic.edu (H. Lin).
[1] Tel.: +1 312 413 3148.

0141-9331/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.micpro.2009.09.002

paths is something else, which is highly unlikely event. To exploit the false positives in the Bloom filter to launch a successful attack, the attacker has to construct two proper values that conform to the false positive pattern, one for the address pointed by the program counter (PC) of a branch instruction for execution that the attacker can succeed to modify, and the other for the target address where the attacker can succeed to inject a malicious code or a trampoline to it. Also note that devising a false positive pattern in the Bloom filter is akin to breaking encryption code and that it becomes even more difficult if we add more control path information to be validated.

Along with the size of the bit vector relative to the size of the set to be filtered, different hash functions adopted in the Bloom filter will cause the access delay, areas, and power consumption of the IBVU will vary. This paper goes over design options of the IBVU in VLSI design. In order to meet the different micro-architecture requirement, two organizations, namely the cascading style and the iterating style, for trade off in time and space in VLSI design are presented. Finally, based on the VLSI design experiment data, the performance degradation inside the processor core is provided.

The rest of this paper is organized as follows: Section 2 introduces the basic idea of how to use the indirect branches' signature to detect the attacks, and Section 3 presents the design and implementation of the Indirect Branch Validation Unit. In Section 4, we consider the effect of the Bloom filter on the validation and based on the experiment data provided in Section 3. We also consider how the insertion of the IBVU will affect the performance of the processor core in this section. In Section 5, we provide the related works on the popular protection mechanism proposed so far and conclude in Section 6.

2. Protecting control flow transfer

2.1. Validating indirect branches

A natural solution to prevent the control data attack is to monitor program execution to ensure that it conforms to a pre-defined specification of its intended behavior [13]. Most researches try to achieve this goal by using a model-based solution to monitor indirect events such as system call sequence [10,11,23]. However, extracting the exact static information is very hard and incurs tremendous space or run-time overhead, which makes the methods not suitable for performance critical and storage sensitive environment. Rather than finding a solution from a system-call level or more general, the level in the application source code, our architectural solution focuses on the behavior of each individual instruction at run-time.

Our scheme is based on an observation that the number of the targets of indirect branches in a program that are actually used is moderate and limited. In our simulation, we run the SPEC2000 benchmarks (using reference inputs) and four server-type applications tested in the real word and record the target addresses occurred in the program. Table 1 shows that most applications do have a moderate number of the targets addresses from indirect branches. This is not surprising because a program typically executes a limited number of loops and has confined number of dynamic execution paths in terms of control flow transferred by indirect branches. For example, return instructions always return to their call sites and indirect calls usually jump to the pre-defined functions or subroutine entrances with the same prototype. On the other hand, indirect jumps have a handful of possible targets, whose number is also limited. Hence, we have proposed to incorporate a hardware table containing all possible legitimate targets for indirect branches [19].

Table 1
Target PC # and IBP # of SPEC 2000 INT benchmarks and four server-type applications.

Benchmark	Target PC #	BPC‖TPC #	Benchmark	Target PC #	BPC‖TPC #
gcc	8218	8592	eon	5980	6396
crafty	2652	3162	gap	2965	4480
apache	2526	6055	***sshd***	2016	5593
ftpd	1224	3003	***telnetd***	1127	2247

We validate the target of the given indirect branch instance, which is called TPC (Target PC). Such a validation can prevent the control flow from jumping to the implanted code and/or impossible targets. However, an adversary might possibly utilize a legitimate target to perform malicious operations such as in return-to-libc attacks. Thus as a remedy, the branch's PC, which is called as BPC, can be added into hardware table when validating. Pairing BPC and TPC together, denoted as BPC‖TPC (branch's PC and target PC), and call it an indirect branch pair (IBP), we effectively apply a new constraint on each indirect branch such that any control flow transfers invoked by it is guaranteed to be from an intended branch site to a legitimate destination. Our validation unit is effective against a wide range of attacks and it can be reinforced even more by including addresses of conditional branches up to the indirect branch instance into the hardware table. Our most recent work shows that adding the branch history as dynamic behavior signature of the indirect branch instance does not cause any difficulty except moderate increase in space requirement [18].

2.2. Training IBP table

In addition to what to include in the hardware table, how to fill it up with the legitimate IBPs is another challenging problem. One way is to go through the static or run-time analysis, in which the legitimate IBPs are extracted from the existing execution trace of legacy code offline or during the software testing and development phase. The linker and loader can also help to find legitimate targets of branches when they patch the program with absolute addresses.

Alternatively, the method we are using in this paper is the online training, a widely adopted method in many behavior-based protections [5,9,10,12,17,19,26]. That is, we train the application in a particular time frame, or until the number of interested IBP signatures converges. We test our IBP convergence on Red Hat Linux 7.3 over Bochs-2.2.6 [3], a full-system Intel Pentium emulator. The Linux kernel is modified so that the hardware emulator becomes aware of the process information, such as TPC and BPC information of the interested indirect branch instances. We experiment four server-type applications, *sshd*, *telnetd*, *ftpd* and *apache* in the real world. We record the number of the IBPs against the number of the indirect branches that have been executed. Fig. 1 shows that the number of IBPs does converge quickly, i.e., no new additional IBPs can be found. This confirms a common sense that the program behavior space is fairly limited while its input data space may be unlimited. Profiling of the four server programs shows that for each indirect branch instance there can be at most about 64 K different dynamic execution paths, i.e., up to 16 different conditional branches in static program code, reaching to it from the previous indirect branch instance. However, we have observed that the IBP set collected with random input data patterns exhausting all possible execution paths changes little after a certain period of initial training period (of from a few hours for sshd to a couple of weeks for apache), which means that the 16 different conditional branch instructions do not make independent branch decisions, resulting much less dynamic execution paths than the 64 K paths possible in theory.

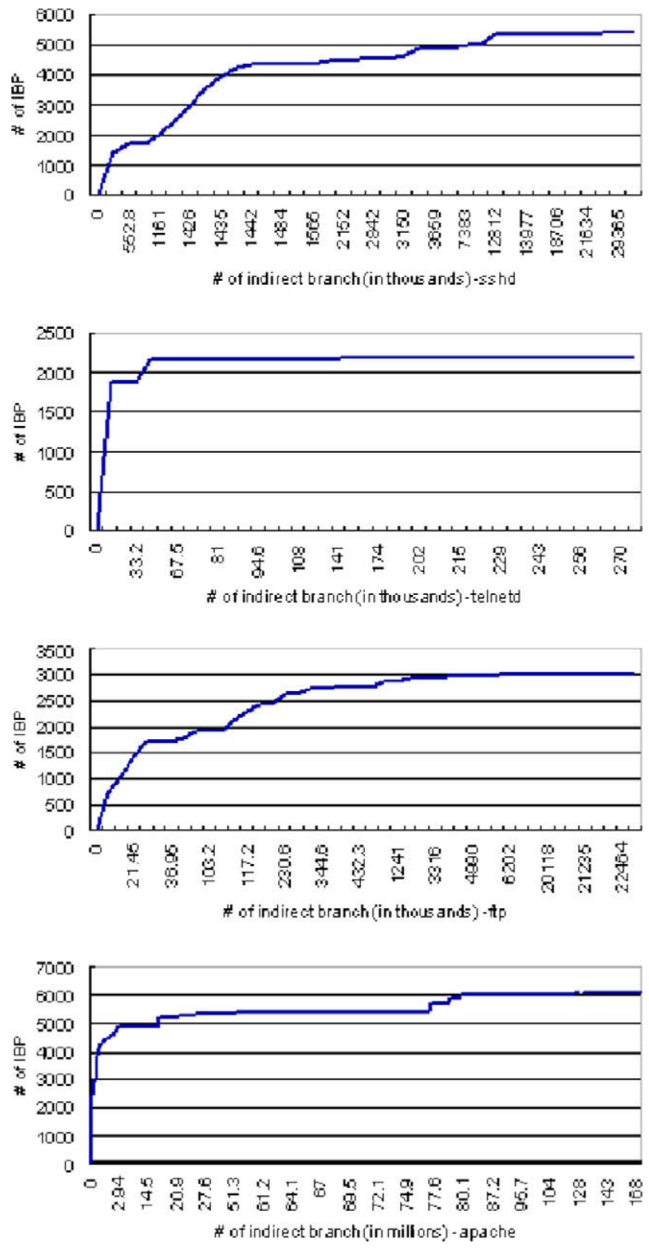

Fig. 1. The convergence speed of IBPs numbers for sshd, telnetd, ftp and apache.

2.3. Validating IBPs only on mis-prediction

After the training period, the IBVU compares each encountered BPC||TPC with the legitimate signatures that have been collected in the hardware table during the run-time. The hardware table for the legitimate IBP set can be represented in various ways depending on speed and storage limitation, a simple bit vector to the whole table implemented in hardware or software. One interesting observation we have made is that many processors are already doing a portion of validation in the form of indirect branch prediction. Also, the significant portion of the hardware table for the IBP set is already implemented in processor core. Modern processors typically incorporate hardware components, e.g. branch target buffer (BTB) and return address stack (RAS), to do branch prediction. These components are initialized to be empty and gradually filled up with targets that have been used at run-time. Since the IBVU checks every target before it is loaded into the program counter, the targets presented in BTB and RAS must have passed the validation in the first place. This implies a control flow transfer on a correct branch prediction is guaranteed to be safe. On the other hand, during an attack, the target address in memory is corrupted and will not match the validated one in the RAS or BTB, resulting in a mis-prediction. Notice that while an attacker is able to overwrite a value in memory due to all kinds of vulnerabilities, it cannot directly compromise the content in the software-transparent prediction units at the same time. Consequently, a mis-prediction event of an indirect branch becomes a symptom of an attack. Hence, the validation can be activated only on that event, rather than every instance of the indirect branch.

3. Implementing Indirect Branch Validation Unit

Based on the data shown in Table 1, we can see that the IBVU should have the following properties: (1) The number of IBPs ranges significantly between programs, thereby an adaptive scheme is desirable to minimize the search time and power consumption. (2) As the maximum IBPs are wildly varying from program to program, a PC-index table or a flat IBP table are not preferred. (3) The fixed-sized hardware table must be able to handle the overflow problem properly. Discarding legitimate IBPs on overflow is undesirable because re-collecting the IBPs at run-time is difficult and expensive. Based on these observations, we propose to utilize a Bloom filter to accommodate and validate the IBPs.

3.1. Bloom filter basics

A Bloom filter [2] is a space-efficient data structure that is used to test whether an element is a member of a set. It tries to answer a query whether in a set $S = \{x_1, x_2, \ldots, x_n\}$ a given element x is included or not without actually storing the elements into the set. The filter is described by a vector of m bits (all initially set to 0) with k independent hash functions with a range of 0 to $m - 1$. The results of the hash functions are used to index the location of the m-bit vector, thus the chosen k locations will give the exact information whether the element is met before or not.

During the insertion or training phase, whenever an indirect branch instance is accessed, the bit value of the IBP pairs becomes the input of the k hash functions. Each return value from the hash function is used to index into the m-bit vector and that bit position is set to '1'. Similarly, during the query or run-time phase, the k locations returned by the hash functions are checked to see if they are already set to '1'. If the bit values in all locations from the hash functions are set, the Bloom filter is said to contain the current BPC||TPC pattern and the indirect branch instance passes the validation.

However, there is a certain chance that the upcoming indirect branch instance may not be included in the set but all its corresponding bits in the vector happen to have been set to '1' by other instances inserted before. This "false positive" is determined by three parameters, namely, the number of hashes k, the size of bit vector m, and the size of the set represented by n. Under the assumption that a hash function selects each bit position with equal probability, the probability of a false positive or false positive rate (FPR) for a Bloom filter can be expressed as [2]

$$(1 - (1 - 1/m)^{kn})^k \approx (1 - e^{-kn/m})^k \tag{1}$$

For a given m/n, Eq. (1) is minimized when

$$k = \ln 2 \times m/n \tag{2}$$

Although the false positives are inherent cost of the Bloom filter for the benefit of a simple time-and-space efficient set representation, one should note that the ***false positives from the Bloom filter are NOT directly translated attack vulnerability of escaping the pro-***

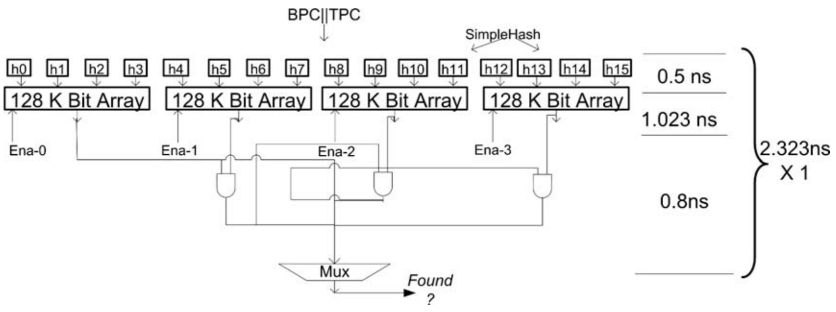

Fig. 2. The validation unit cascading four 128 K-bit Bloom filters.

tection. To exploit the false positives in the Bloom filter to launch a successful attack, the attacker has first to construct two proper values that conform to the false positive pattern, (1) the PC of a branch or memory access instruction that the attacker can succeed to modify and (2) the target address where the attacker can succeed to inject a malicious code or to update, in addition to conforming to a legitimate sequence of conditional branch outcomes reaching to the current PC. Note that our use of Bloom filter necessitates the attacks to break the encryption of k hashes of a basic validation unit, i.e., IBP, which can be made arbitrarily long with additional context information. Such additional context information are readily available in most processor architecture, e.g. local frame pointer ($fp) or the Last Branch Record (LBR) mechanism of the Intel P4 processor. With larger k and longer bit-pattern (for an IBP with added context information), breaking of such an encryption becomes increasingly harder. Another added difficulty for attacker is program structure and memory map of the system; not all bit patterns attacker reversely extracted from Bloom filter can be utilized for attack. From cryptography perspective, extracting false positive patterns may be considered not difficult, but structural limit due to memory map and the additional control flow context information such as branch history makes it extremely difficult, if not impossible, to make a false positive pattern for the Bloom filter a successful attack launch, i.e. a false positive from the program protection perspective.

3.2. The design of the IBVU

In order to incorporate the IBVU into the processor core, the area, power, and performance limitation have to be evaluated carefully. We propose two organizations to utilize Bloom filter to support the validation in IBVU.

The first organization is optimized for access speed at the cost of more hardware resources. We call it a cascading style design in that several smaller bit vectors are used in the Bloom filter and their results are connected in a cascading way. A simple example to describe this organization is provided below. With (1) as the guide, we have chosen $n = 16$ K (indicated from Table 1), $k = 4$ and obtained m of 96 K from (2). Assuming with a 128 K bit vector, a Bloom filer achieves an FPR of 0.024, which may be too high in securing a computer system. Therefore, four 128 K Bloom filters with independent hash functions is cascaded as shown in Fig. 2. When enabling all four 128 K-bit Bloom filters, one IBP must be validated by all four independent Bloom filters, resulting in an overall FPR of roughly the products of each basic unit's FPRs, i.e. $0.0244 = 3.32\text{E}-7$, which is roughly equivalent to breaking a 48-bit DES encryption. The actual FPRs for real server-type applications are even lower. For example, apache, with roughly 6100 IBPs, has a FPR of 6.08E−13. Also, as mentioned earlier, note that this FPR is *not* a rate for false positive in validating an IBP but rather just a rate for false positive in the Bloom filter. To have a false positive in the Bloom filter a false positive in validating an IBP, the false positive pattern must be of the addresses conforming to the system and the attacker's way.

Alternatively, an iterating style organization may be devised with less hardware cost but longer access latency. This design consists of only one larger bit vector of 256 K and one hash circuitry as shown in Fig. 3. To maintain a similar FPR, the hashing and bit vector access are performed multiple times sequentially (effectively increases k). And the outputs are buffered and ANDed to determine the final result. Assuming the Bloom filter is accessed twice ($k = 8$) and three times ($k = 12$) with 6100 IBPs as in apache, the corresponding FPRs are 8.27E−8 and 5.6E−9, respectively.

Depending on different targeted FPR requirements, the size of the bit vector in the Bloom filter can be varied. Table 2 shows the delay, area and energy consumption of the bit vector with a size between 64 k bits and 512 k bits using CACTI [25] with the 90 nm technology.

3.3. Hash function implementation

FPR in this context means the Bloom filter might report a non-existing IBP as a legitimate IBP, allowing a possible attack to pass the checking without being detected. A false positive of a Bloom filter is translated into a false negative of the system. Therefore we need to carefully select the filter parameters, i.e. m, k, n, and the hash functions, to minimize FPR within hardware budget while

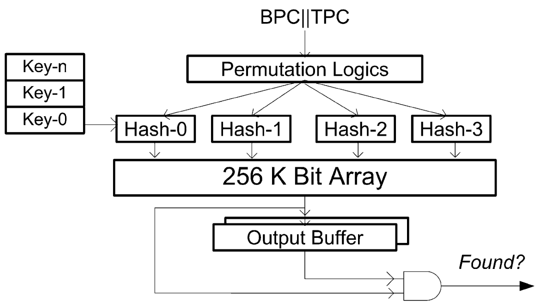

Fig. 3. The validation unit using one 256 K bit Bloom filter.

Table 2
Access delay, energy consumption and area cost of various bit vector size using CACTI under 90 nm technology.

Bit array (bits)	Access delay (ns)	Energy consumption (nJ)	Area (mm^2)
64 K	0.785	1.0906	1.1719
128 K	1.023	1.4329	1.8389
256 K	1.062	1.9267	3.0181
512 K	1.7781	2.7646	14.4384

not hampering performance too much. The value of the filter parameters is strongly related to the characteristics of the trained program. However, the high quality of hash function, which is usually independent of the program being trained, warrants a low FPR close to the ideal value. In this section, we propose three different implementations of the hash functions. Their hardware cost and the resultant simulated FPR are also presented.

First mechanism is to make use of the existing hash/encryption components if the processor has already integrated ones to support trust computing, such as SHA-1 circuitries [21]. It is reported the delays for SHA-1 in 180 nm technologies are around in the order of 100 ns. This mechanism, making the best use of the resource sharing, saves the cost of designing and adding new component in the processor core.

Alternatively, we make use of the universal classes of hash functions [6]. Any class of functions that is universal$_2$ has the property that given any sample input, a randomly chosen member from this class will be expected to distribute the sample evenly, thus achieve the analytically predicted performance of the hashing function. H3 is a hardware-friendly hash scheme has the property of universal$_2$. Ramakrishna et al. [16] have shown experimentally that it can achieve analytically predicted performance. In H3, each hash consists of a randomly generated $i \times j$ matrix Q in order to hash an i-bit element, X, into a j-bit address, Y. Hashing is computed as below:

$$\mathrm{Hash}(X) = Y = \{X_0 \text{ AND } Q(0)\} \oplus \{X_1 \text{ AND } Q(1)\} \oplus \cdots \oplus \{X_i \text{ AND } Q(i)\},$$

where the X_k is the kth bit of X, $Q(k)$ is the kth row of Q.

This simple expression implies an efficient structure that can be implemented in hardware as shown in Fig. 4. Matrix Qs are typically small (e.g. 64 × 17 bits each assuming m = 17 (128 K bit vector)) and can be implemented as ROMs. Table 3 shows the delay of H3 function and area and power of both the hashing and each Q matrix.

At last, if one tries to minimize the hardware overhead and validation latency at the cost of a relatively high FPR, an easy-implement hash scheme, named *Simplehash* can be adopted. It does hashing by shuffling and xor-ing the bits in the 32-bit PC and the 32-bit target address as shown in Fig. 5. The right most $\log_2 m$-bits portion of the result are maintained as the index to the bit vector. This method neither requires complicated algorithm nor depends on an additional random inputs. Our synthetic result shows that it only incurs 0.5 ns delays, however, with about a magnitude larger FPR in the worst case.

In order to compare the validation performance, we simulate on the FPR of these hash implementations. From Table 1, we can see that gcc has the most IBPs (~10,000) in SPEC2000 and assume in the worst case an application has 16 K IBPs (or n = 16 K) in our experiment. Thus we choose the max member of n equals to 16 K. The vector length varies from 128 to 512 K bits, which is a pretty moderate hardware budget for today's processor. And the number of hash functions ranges from 4 to 8 according to the micro-architecture design experiences.

The simulation results of the FPR for different hash mechanisms are shown in Table 4. For the comparison purpose, we also provide in the table an ideal FPR calculated from (1). From Table 4, we only provide the FPR for the single Bloom Filter unit, since there are two

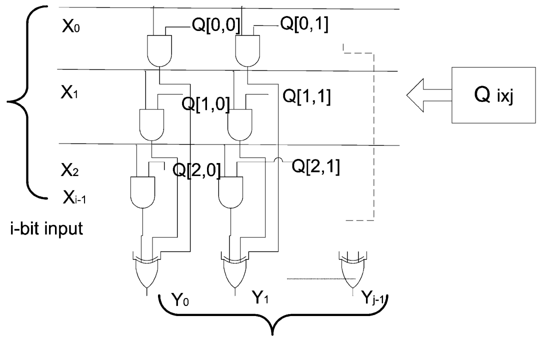

Fig. 4. One H3 hash unit with i-bit input and j-bit output.

Table 3
Synthetic results for H3 hash function (a) and one Q matrix implemented in RAM (b) under 90 nm technology.

m/bit array size	Access delay	Power (mW)	Area (nm^2)
(a)			
17/128 K	1.48 ns	3.40	9560
18/256 K		3.61	10122
Q size (bit)		Power (mW)	Area (nm^2)
(b)			
64 × 17		3.359	47790
64 × 18		3.592	50612

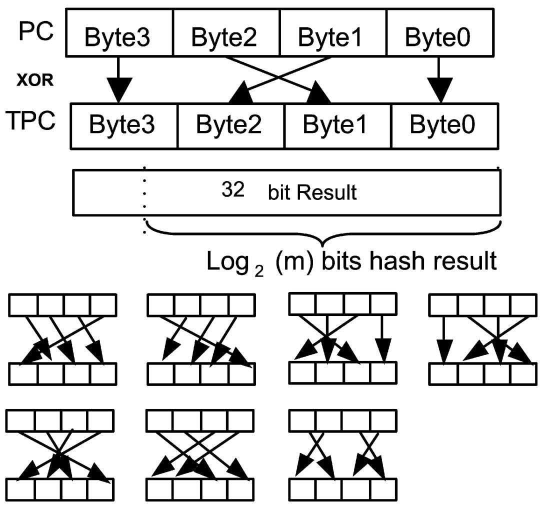

Fig. 5. One of the Simplehash functions as well as other 7 hashing functions with different permutation.

Table 4
Respective FPRs when ideal hash functions and other three implementations are employed.

Bloom filter parameter			FPR for each unit			
n	m	k	Simplehash	Sha-1	H3	Ideal
16 K	128 K	4	2.62E−2	2.58E−2	2.41E−2	2.396E−2
		5	2.49E−2	2.41E−2	2.20E−2	2.17E−2
		6	2.41E−2	2.36E−2	2.20E−2	2.16E−2
		7	2.75E−2	2.64E−2	2.35E−2	2.29E−2
		8	3.18E−2	3.04E−2	2.57E−2	2.55E−2
	256 K	4	2.65E−3	2.60E−3	2.45E−3	2.394E−3
		5	1.73E−3	1.66E−3	1.41E−3	1.39E−3
		6	1.17E−3	1.13E−3	9.79E−4	9.35E−4
		7	9.88E−4	9.47E−4	7.35E−4	7.015E−4
		8	1.003E−3	9.04E−4	5.58E−4	5.745E−4
	512 K	4	2.10E−4	2.10E−4	1.82E−4	1.906E−4
		5	8.297E−5	6.96E−5	6.94E−5	6.33E−5
		6	3.815E−5	2.86E−5	2.62E−5	2.497E−5
		7	1.764E−5	1.38E−5	1.454E−5	1.13E−5
		8	5.007E−6	5.007E−6	7.39E−6	5.73E−6

organizations being used and the overall FPR for the cascaded or iterated ones can be easily calculated. On the average, using H3 and SHA-1 will give a better result than the Simplehash. However, Simplehash also has its advantage which is the simplicity in the implementation. Overall, no single hash function will win in all situations, but all of them achieve a moderately close value to the theoretical result.

4. Performance evaluation

This section investigates the performance impact of our IBVU when it is incorporated into the processor's pipeline. In our experiment based on CACTI 3.2 [25] with 90 nm technology, we simulate both the cascaded and iterated organization and select the comparatively suitable hash function for each organization [20]. Using four 128 K Bloom filters and Simplehash in a cascading organization design, the validation of IBVU is estimated to incur 2.323 ns delay. On the other hand, using H3 hashing and one 256 K Bloom filter in an iterating organization design, the IBVU delay is about 3.492 ns for each individual validation unit. Thus the delay will be 7 and 10.5 ns if two and three accesses are needed.

Assuming a processor runs at a clock rate of 2.0 GHz, the access latency of the bit vector in the Bloom Filter ranges will be 6 clock cycles, 7 clock cycles, 14 clock cycles, and 21 clock cycles. We test SPEC2000 benchmarks running in Simplescalar that models an out-of-order 4-issue, 9-stage superscalar processor. No special design for indirect branch prediction is assumed and a conservative configuration of BTB and RAS is employed in the simulation. The detail parameter values are provided in Table 5.

Fig. 6 shows the resulting performance degradation with different validation unit access time. The increased mis-prediction penalty for indirect branches due to adding a validation unit has, on average, a moderate impact for most benchmarks. On average, the performance degradations are 2.2%, 2.5%, 4.7% and 6.7%, respectively. It is worth noting that our assumption of the validation delay as an extra part of a mis-predicted penalty is actually fairly conservative. Depending on the implementation, we may overlap the validation operation with other mis-prediction recovery procedures such as renaming table restoring. Thus, validation delays can be hidden partially even completely. Furthermore, special designs for the indirect branch predictor can also be incorporated seamlessly. In literature, many works have been done about improving prediction accuracy for indirect branches [7]. Employing a more aggressive predictor can certainly reduce mis-predictions further, resulting in even less performance impact for our Bloom filter validation.

Table 5
Architecture parameter.

Parameter	Value
BTB	512 set, 4-way set-associative
RAS	8 entries
Branch miss penalty	6 cc, 7 cc, 14 cc, 21 cc
Pipeline stage	9
Branch Predictor	g-share, 12 bits history, 2048 entries
Fetch/dispatch/issue width	4
Instruction window	128 entries
Load/Store Queue	64 entries
I-cache	64 K, 2 way set-asso., 2-cycle hit time
D-cache	64 K, 4 way set-asso., 2-cycle hit time
L2-cache	Unified, 512 KB, 4 way set-associative
L2 access time	10 cycles
Memory	100 cycles access time, 2 memory ports
Function unit	4 Int ALUs, 1 Int MUL/DIV, 4 FPAdder, 1 FP MUL/DIV

5. Related works

Through the years, many contributions, both in software and hardware mechanism, have been made on the securing program execution by restraining the control flow transfers. At the very early time, Data Mark Machine by Fenton [8] enhances every memory word with a tag. Based on the underlying security policy, the tag can be set and checked for potential security violation. As simple variations of the Data Mark Machine, there have been proposals to have a single bit tag attached to each datum to tell if it is from a trusted channel or an un-trusted one [5,10]. Recently, the "data mark" mechanism is evolved into the methods applying the encryption on the control data. The control data value is hidden under the encryption until the last minute of its use for control flow change in program execution [15,22] However, it is impossible to track interested control data accurately and efficiently, and these methods are also not expandable to the future attacks on which the sensitive control data is based is unknown unless all potential control data are encrypted properly.

On the other hand, there are also a large number of the works focusing on the behavior-based protection. What to choose to represent the behavior and how to collect them are two key points. Forrest et al. first observes that short-term system call sequence could be viewed as "a sense of self" for a program [10]. Wagner et al. [23] propose a static analysis method with pushdown automata to build an intrusion model. Besides to choosing the system calls as the behavior, Sekar's finite state automaton proposal associates the system call with the PC [17]. VtPath [9] is an effort that adopts the return address sequence as the program behavior. And Gao et al. [12] extend this work by including execution graph to infer sequence of calls, returns and intra-procedural transition. All the methods above work at the granularity of system calls and may miss the attacks that alter program behavior between system calls [24,26]. In addition, severe performance overhead and slower response time have been reported [22,23] as the system call based protections are often implemented in software due to their way of

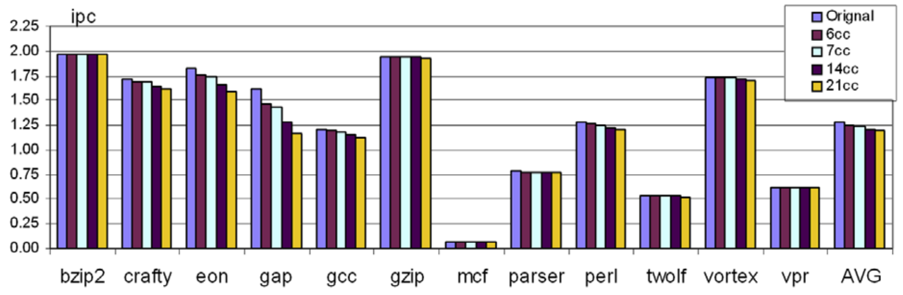

Fig. 6. Performance degradation after validation unit is incorporated. The IPCs are shown for four validation delays, respectively.

collecting program behavior signature. Instead, our architectural scheme, as a complement to system call based solutions, works at instruction level and may detect an attack earlier with moderate performance degradation.

There are also many protection methods in the instruction level. Program shepherding [14] makes use of the binary code interpreter to constrain the control flow transfer when constructing execution traces. However, this method is very complex and introduces non-trivial performance/memory overhead. CFI [1] is a software solution that marks and validates targets for indirect calls and returns through binary rewriting. Their work mainly addresses indirect calls detected statically and mentions little about indirect jumps and their dynamic instances. N-jump [26] by Zhang et al. also works on machine-code level and is based on a similar observation about branch behaviors. Their work mainly focuses on direct branches and treats the indirect branches as special cases. Comparing to their work, our detection object is more specific and including the indirect branch pair is more accurate in representing the program behavior. And our scheme can accommodate the extra information for the behavior signature easily, e.g. branch history up to the indirect branch instance can be included as the basic signature to be hashed into the Bloom filter to make the protection more complete and accurate [18]. The introduction of the Bloom filter to do the validation provides flexibility of adding more information to the behavior signature without incurring serious performance, area and power overhead.

6. Conclusion

Current processor architecture is vulnerable for attacks that are aimed at altering control data, the data that are used as the target addresses of the indirect branches. The current systems lack the validation mechanism to check the legitimacy of the branch and its target address. We have presented a micro-architectural enhancement, called Indirect Branch Validation Unit (IBVU), to do the validation of an indirect branch's PC and its target, (named IBP). All legitimate IBPs allowed under a given security policy will be collected and stored in a hardware table implemented in the Bloom filter first and then the IBVU validates any control flow transfer at run-time at the mis-prediction of the indirect branch. Our proposal makes the best use of the branch prediction mechanism in the current processor and does the validation only necessary after the mis-prediction happens.

We also consider the Bloom filter design for the IBVU to record the IBP information of the indirect branch instance without actually storing it. However, to largely save the storing space and performance cost, the Bloom filter will introduce the false positive rate (FPR) during the training period. The FPR is closely related to the design parameters of the Bloom filter and the hash function used. In order to restrict the values of the parameters under the area and power requirement, we design IBVU in two organizations by cascading smaller Bloom filter units or iterating them. On the other hand, to analyze the effect of hash quality on the FPR, we have tested three different commonly used hash mechanisms and have found that the results, although varied in situations, achieve the desired FPR at moderate time, area and power penalty. Moreover, the validation mechanism only results in small performance degradation when incorporated in the processor pipeline. Our simulation shows little performance penalty even in our conservative simulation environment. A processor core with secure control flow creates a basis for designing and building more trusted devices and cyber infrastructure.

One potential limitation of validating an IBP against a pre-collected legitimate IBP set is handling of never encountered before but a legitimate IBP instance at run-time. This case needs to trigger a post-mortem analysis to figure out if it may be a false negative when the IBVU raises an exception signal. A static control flow graph may help to expedite the post-mortem analysis. Our future work will include how to do the post-mortem analysis along with updating the legitimate IBP set per the analysis results.

Acknowledgements

We acknowledge the help of Arnold Lee who did Verilog description and synthesis of the validation unit. This work was supported in part by a special research fund from the college of information and communications at the Korea University, Seoul, Korea, the IT R&D program of MKE/IITA [2009-F-039-01 Trustworthy Computing], and a funding from the US National Science Foundation (CNS-0627431).

References

[1] M. Abadi, M. Budiu, U. Erlingsson, J. Ligatti, Control_flow Integrity, in: ACM CSS05, November 2005.
[2] B. Bloom, Space/time tradeoffs in hash coding with allowable errors, Communications of the ACM 13 (1970) 7.
[3] Bochs, The Open Source IA-32 Emulation Project, <http://bochs.sourceforge.net>.
[4] CERT Security Advisories, <http://www.cert.org/advisories/>.
[5] J. Crandall, F. Chong, Minos: control data attack prevention orthogonal to memory model, in: Proceedings of the 37th International Symposium on Microarchitecture, December 2004.
[6] L. Carter, M. Wegman, Universal classes of hashing functions, Journal of Computer and System Science 18 (2) (1979).
[7] P. Chang, E Hao, Y. Patt, Target prediction for indirect branches, in: Proceedings of the 24th ISCA, 1997.
[8] J. Fenton, Memoryless subsystems, Computer Journal 17 (2) (1974) 143–147.
[9] H. Feng, O. Kolesnikov, P. Fogla, W. Lee, W. Gong, Anomaly detection using call stack information, IEEE Symposium on Security and Privacy, May 2003.
[10] S. Forrest, S. Hofmeyr, A. Somayajo, T. Longstaff, A sense of self for unix processes, in: Proceedings of the 2000 IEEE Symposium on Security and Privacy, 1996.
[11] J. Giffin, S. Jha, B. Miller, Efficient context-sensitive intrusion detection, in: 11th Annual Network and Distributed Systems Security Symposium, February 2004.
[12] D. Gao, M. Reiter, D. Song, Gray-box extraction of execution graphs for anomaly detection, in: The ACM CCS Conference, 2004.
[13] C. Ko, C. Fink, K. Levitt, Automated detection of vulnerabilities in privileged program s by execution monitoring, in: Proceedings of the 10th Computer Security Applications Conference, 1994.
[14] V. Kiriansky, D. Bruening, S. Amarasinghe, Secure execution via program shepherding, in: Proceedings of the 11th Usenix Security Symposium, 2002.
[15] G. Lee, A. Tyagi, Encoded program counter: self-protection from buffer overflow attacks, Proceedings of the First International Conference on Internet Computing, June 2000.
[16] M. Ramakrishna, E. Fu, E. Bahcekapili, A performance study of hashing functions for hardware applications, in: Proceedings of the International Conference on Computing and Information, 1994.
[17] R. Sekar, M. Bendre, P. Bollineni, D. Dhurjati, A fast automaton-based method for detecting anomalous program behaviors, in: Proceedings of IEEE Symposium on Security and Privacy, 2001.
[18] Y. Shi, G. Lee, Augmenting branch predictor to secure program execution, in: The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2007, 25–28 June 2007, Edinburgh, UK.
[19] Y. Shi, S. Dempsey, G. Lee, Architectural support for run-time validation of control flow transfer, in: Proceedings of the IEEE International Conference on Computer Design (ICCD06), October 2006.
[20] Y. Shi, A. Lee, G. Lee, T.-J. Park, T.-C. Jung, B.-C. Kang, Indirect branch validation unit for secure program execution, in: The 10th IEEE Symposium on Low Power High Speed Chips (Cool Chips X), Yokohama, Japan, April 2007.
[21] SHA-1, <http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf>.
[22] N. Tuck, B. Calder, G. Varghese, Hardware and binary modification support for code pointer protection from buffer overflow, in: Proceedings of the 37th International Symposium on Microarchitecture, 2004.
[23] D. Wagner, D. Dean, Intrusion detection via static analysis, in: Proceedings of the IEEE Symposium on Security and Privacy, 2001.
[24] D. Wagner, P. Soto, Mimicry attack on host-based intrusion detection system, in: ACM CSS 02, November 2002.
[25] S. Wilton, N. Jouppi, CACTI: an enhanced cache access and cycle time model, in: IEEE JSSC, vol. 31(5), May 1996.
[26] T. Zhang, X. Zhuang, W. Lee, S. Pande, Anomalous path detection with hardware support, in: Proceedings of CASES, 2005.

Gyungho Lee received the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign in 1986. He is currently a professor in the college of Information and Telecommunications, Korea University, Seoul, Korea. His research and teaching interests are in computer architecture, microprocessor design, and computer/network system security. He was an IEEE Computer Society Distinguished Visitor from January 2000 to December 2002. He has been an editor for several journals including the IEEE Transactions on Parallel and Distributed Systems. He is elected to a Fellow of the American Association for the Advancement of Science (AAAS) in 2006 and awarded a "University Scholar" designation, the highest honor bestowed by the University of Illinois upon its faculty, in 2007.

Yixin Shi received his B.S. and M.S. degrees in Electronics Engineering both from the Shanghai Jiao Tong University in 1997 and 2000, respectively. He received the Ph.D. degree in the Department of Electrical and Computer Engineering, University of Illinois at Chicago in 2007. He is currently a software engineer in platform group at Google Inc. He is interested in architectural support for security, cache design in wide-issue processors, architectural simulator.

Hui Lin is a Ph.D. student in the Department of Electrical and Computer Engineering, University of Illinois at Chicago. His research area includes microprocessor architecture and trusted computing.