Architectural Support for Run-Time Validation of Control Flow Transfer

Yixin Shi, Sean Dempsey, Gyungho Lee

Department of Electrical and Computer Engineering
University of Illinois at Chicago, Chicago, IL 60607
{yshi7, sdemp1}@uic.edu, ghlee@ece.uic.edu

***Abstract*—Current micro-architecture blindly uses the address in the program counter to fetch and execute instructions without validating its legitimacy. Whenever this blind-folded instruction sequencing is not properly addressed at a higher level by system, it becomes a vulnerability of control data attacks, today's dominant and most critical security threats. To remedy it, this paper proposes a micro-architectural mechanism to validate control flow transfer at run-time at machine instruction level. It is proposed to have a hardware table consisting of legitimate indirect branches and their target pairs (IBPs) to aid the validation. The IBP table is implemented in the form of a cascading Bloom filter to store the security information as well as to enable fast validating. Based on a key observation that branch prediction unit existing in most speculative-execution processors already provides a portion of the control flow validation, our scheme activates the validation only on indirect branch mis-predictions. Because of the Bloom filter and the rarity of mis-predictions of indirect branches, the validation incurs moderate storage overhead and little performance penalty.**

I. INTRODUCTION

Exploiting program vulnerabilities becomes great threats to modern information infrastructure. Under the current software and hardware interface, most malicious attacks try to take over the control of a victim computer system by changing the program control flow. This can be done only by compromising the control data to re-point the processor's program counter (PC) to the attacker's way. Control data are the data that could be loaded into the PC and can be dynamically generated and changed at run-time. It includes the return addresses in the stack, the function pointer variables, and special data structures for non-local jumps (e.g. setjump/longjmp buffer), etc. Attackers exploit the vulnerabilities such as buffer overflow[18], format string vulnerability[19], heap corruption, and double free bugs [17], integer overflow[3] to change the control data. With the control data altered to the attacker's way, the attacker can perform any operations that the victim program has permission to do. Reports from CERT[6] show that control data attacks are dominant and the most critical security threats today.

Control data attacks typically break "normal" control flow but still follow the instructions' semantics without explicitly violating any security polices. This makes traditional measures such as access control or data/code encryption alone hard to prevent the control data attacks. Recently, hardware-based schemes that track and protect the control data directly are proposed[7][9][15][22][23]. However, identifying and tracking control data are generally difficult to implement correctly due to their dynamic nature and aliasing. Many model-based anomaly detection approaches[10][11] are suggested also. These methods use some specific run-time information, e.g. system call sequence, to approximate the program behavior indirectly. Their effectiveness, however, often suffers from the inaccurate information being monitored and large performance and/or memory overhead.

This paper introduces a novel idea of protecting program control flow at micro-architecture. We propose to validate every control flow transfer in the instructional level at the moment a taken branch is about to update the program counter. We focus the validation range down into each indirect branch because only the targets of indirect branches can be dynamically changed and can be potentially overwritten through exploitations from an attacker. A set of legitimate indirect branches and their targets stored in a hardware table is employed to validate each control flow transfer. To overcome the overflow problem caused by the limited capacity of a hardware table as well as enable fast validation, a Bloom filter storing the security information is proposed to be incorporated into the processor's pipeline. We further detail a validation unit that cascades several small Bloom filters to minimize the inherent false positive rate. Finally, we find that the validation can safely be activated only on a branch mis-prediction in architecture, resulting in lower performance degradation than otherwise possible

The next section describes the motivation and our validation method on control flow transfer. Section 3 shows how to design a Bloom filter in a cascading fashion to facilitate the protection. Section 4 illustrates the way to reduce validation times and depicts the architectural modifications. Section 5 evaluates the performance impact. Section 6 presents related works and a comparison of our scheme to existing security solutions and we conclude the paper in section 7.

II. PROTECTING CONTROL FLOW TRANSFER

To successfully launch a control data attack, an attacker has to 1) inject malicious inputs, 2) overwrite the control data by exploiting buffer overflow bugs or format string vulnerabilities or through other means, and 3) execute the tampered control flow transfer instructions and eventually run the malicious code. Many measures have been proposed to intercept the attack in each of the steps above. To stop the attacks in step 1 or 2, one has to identify, protect, and track every control datum perfectly, which is very difficult, if not impossible. Instead, we propose to insert a checking and validation mechanism at micro-architectural level in step 3 for every control flow transfer instruction.

A. *Control Flow Graph and Control Data Attack Scenarios*

An example program that is vulnerable to control data attacks and its control flow graph (CFG) are shown in Fig-1. The nodes representing in ◎ are indirect branches, which could be a return (e.g. *I1* and *I2*) or a non-return indirect jump (e.g. *I3*). Assume that *f1()* contains a buffer overflow vulnerability and *f4()* has a format string vulnerability while the function pointer variable, *fp*, is also subject to be overwritten maliciously. The control data attacks exploit these vulnerabilities to overwrite the return address or function pointer variables. It is the ultimate executions of the compromised indirect branches that deviate program execution from the normal behavior. As a result, a new node will be inserted and/or a completely new execution path is created in CFG (e.g. I1->X->Y, I2->Y after overwriting return address of *f1* or *f4*). Or a new execution path is generated to bridge two nodes which previously do not have any control flow between them (e.g. I3->Z after compromising the function pointer *fp*).

B. *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[14] or use a model-based solution to monitor other indirect events such as system call sequence[10][11][24]. However, extracting the exact static control flow information is very hard and incurs tremendous space or run-time overhead to be practical. Rather than finding a solution from a high level of application source code or even behavior specification, our architectural solution focuses on each individual instruction at run-time. In the current processors, control flow tracking at machine instruction level is blindfolded without validity check. Processors blindly follow the program counter to fetch and execute instructions. We believe this a fundamental flaw in hardware that causes the endless chase of software vulnerability, its exploitation, and its patch. To rectify this, control flow validation should be done at the machine instruction level.

At instruction level, high-level descriptions of control flow transfers are ultimately translated into direct branches and indirect branches in binary code. The target of a direct branch is typically determined by the compiler and it points to one single location wired in the instruction bits. In practice, there exist, but very rare, cases that an attacker can compromise the decision of a direct branch *and* perform intended malicious operations in the pre-determined execution path in order to hijack the system. Instead, most existing control data attacks[6] seek to compromise the target of an indirect branch. We observed that the target of an indirect branch is often allocated dynamically in the data area of the program's address space, and an attacker can manage to overwrite it by exploiting memory corruptions techniques mentioned in the introduction part. Later execution of the compromised indirect branch gives the attacker great flexibility to re-point the control flow to *any* appropriate location he desires (e.g. the entry address of *exec()* or *system()* in Fig-1). Hence, any control flow transfer by an indirect jump should be validated before the architecture actually uses it.

To define a control flow transfer by an indirect branch in CFG, we are concerned with two pieces of information. One is the location of the indirect branch or the branch site (BPC) and the other is the target address (TPC). A validation of both the branch site and the target ensures that any indirect branch is always from a legitimate branch site and to a legitimate destination. This is necessary because such a validation not only prevents control flow from transferring to an unintended

```
int main(argc, argv)
{
  int (*fp)();
  char buf[];
  if (…) { f1(); fp = f4;}
  else
    {f2();f3(); f1(); fp= f5;}
  strcpy(buf,…);
   (*fp)(argv);
   if () else();
   system();
}

f1() {
char buf[];
strcpy(buf,argv..);
}

f4() {
printf(argv);
};
```

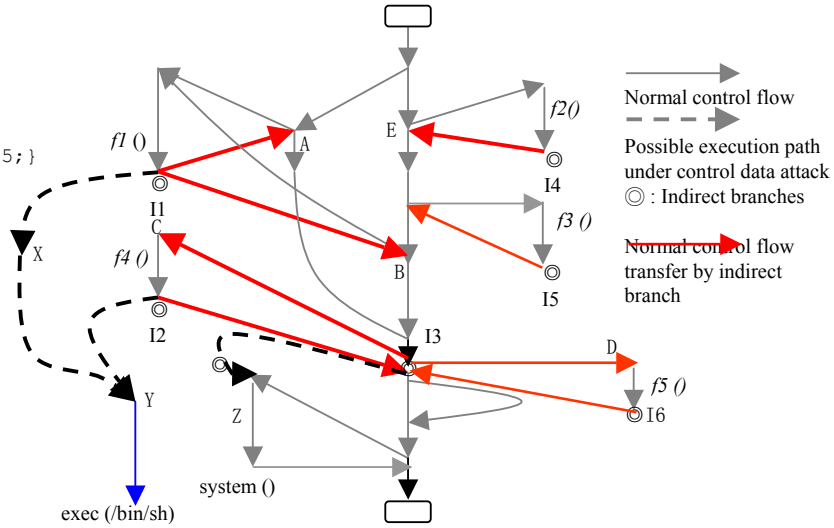

Fig-1. A vulnerable program with possible attacking scenarios.

destination but also enables intercepting jumps heading to a legitimate target but from an illegal source site (e.g. return-to-libc attacks). We pair BPC and TPC together, denoted as BPC||TPC (branch's PC and target PC), and call it *an Indirect Branch Pair* (*IBP*). Each indirect branch can have multiple targets thereby may have more than one IBP. By introducing the IBPs, we effectively apply a new constraint on the targets of each indirect branch such that it can only use a limited number of validated values. This greatly reduces the chance that an attacker can redirect the control flow to a new execution path that originally does not exist in the program.

The control flow validation is performed by checking against an *IBP table,* proposed as a hardware component for a collection of all the legitimate IBPs in a program, at each indirect branch. It represents "normal" behavior and provides a reference to check if software starts to behave abnormally. At run-time, any encountered IBP that fails to match any of the target addresses in the IBP table will cause the processor to raise a hardware exception. The exception is captured by the operating system (OS) and the OS may simply halt the execution and issue an alert to the administrator to take further actions. For example, the CFG in Fig-1 has a legitimate IBP table of {*I1||A, I1||B, I2||G, I6||G, I3||C, I3||D, I4||E, I5||F*}. Suppose due to a control data attack, a control flow transfer from I1 to X is invoked. A validation on the corresponding IBP (I1||X) will produce a mis-match and this transfer can be intercepted.

C. Training IBP table

There are basically two ways to fill up the IBP table with legitimate IBPs. One is through static or run-time analysis. We may extract the legitimate IBPs from the existing execution trace of legacy code offline. Or, the linker and loader can help find legitimate targets of branches when they patch the program with absolute addresses. Also, during the software testing and development phase, the test cases being used should cover most, if not all, possible execution paths for each branch; therefore an IBP table can be generated as a side product.

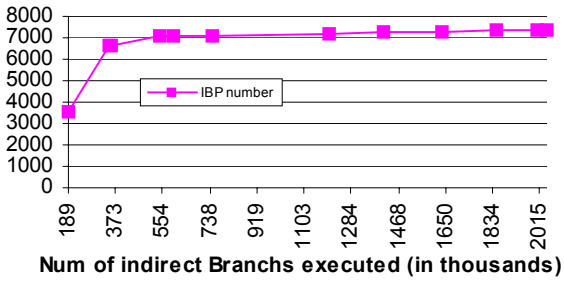

Fig-2. The number of unique IBPs against indirect branches that have been executed. Each data point is an additional workload.

The second way to initialize the IBP table is to perform "training" as many model-based solutions have done [10][11][25]. By running the application either in a particular time interval or until the unique IBPs converges in a secure environment, the processor can regard all seen IBPs as legitimate ones. We test IBP convergence of an *Apache* server on Red Hat Linux 7.3 over Simics[16], an IA-32 emulator. We generate both static and dynamic loads from a remote machines while colleting the addresses of the indirect branches and targets on the simulated machine. Fig-2 shows that the number of IBPs does converge quickly.

III. The IBP Table Design using Bloom Filters

A. Analysis of Indirect Branch Characteristics

To study the indirect branch characteristics in applications, we have profiled indirect branches and their targets or IBPs of SPEC2000 benchmarks. The Alpha binaries of the benchmarks with reference inputs are simulated completely in sim-profile, a simulator in Simplescalar[1]. We also studied two real applications, *Apache 1.3.27-8*, and *sshd*, running over Simics. Table-1 shows that the number of IBPs ranges from a several hundred to 10,000. No excessive number of IBPs is observed and most programs have a modest number of IBPs around 1,000 except the programs that include many recursive functions calls.

TABLE-1: UNIQUE INDIRECT BRANCH AND TARGET PAIR (IBP) IN SPEC2000 BENCHMARKS AS WELL AS TWO OTHER APPLICATIONS.

SPEC2000 benchmarks/applications with IBP #							
gcc	9624	twolf	1289	mgrid	455	Apache init	4844
perlbmk	4295	vpr	1262	applu	422	main	7325
vortex	3735	apsi	1039	lucas	385	sshd	3656
eon	2416	galgel	964	equake	373		
sixtrack	2289	facerec	747	gzip	288		
fma3d	1891	wupwise	585	mcf	287		
crafty	1675	mesa	547	bzip2	286		
gap	1599	ammp	518	art	245		
parser	1434	swim	469				

We have also measured the maximum number of targets one indirect branch can have for all benchmarks in SPEC2000. This metric reflects how many values one single indirect branch may take as its target.

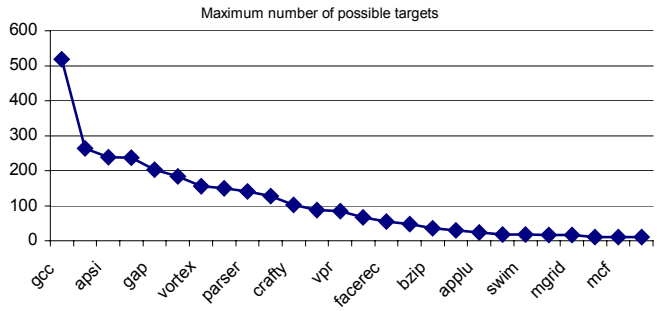

Fig-3. The maximum number of possible targets per indirect branch of SPEC2000 benchmarks.

Fig-3 shows they vary drastically from one benchmark to another. For example, the maximum number of targets per branch in *lucas* is only 10, while some indirect branches in other benchmarks like *gcc*, *sixtrack*, *eon*, and *apsi* can have more than 200 possible targets.

B. Observations

Based on the data shown above, we conclude that the IBP

table 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.

C. 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 (initially set to 0) with k independent hash functions with a range of *0* to *m-1*.

During the initialization phase, k hash functions are applied to the input element. Each return value from the hash function is used as an *index* to the m-bit vector and that bit *position* is set to 1. During the query, 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 1, then the Bloom filter is said to contain the pattern.

There is a certain chance that x may not be in the set but all its corresponding bits in the vector happen to have been set to 1 by other elements 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 n. The probability of a false positive or false positive rate (*FPR*) for a Bloom filter is

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

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

$$k = \ln 2 \times m/n \quad . \quad (2)$$

D. Storing IBPs into a Bloom filter

Three features of the Bloom filter make it an ideal hardware implementation to store the security information. First, while risking false positives, Bloom filter has a strong space advantage over other data structures for representing sets due to its compactness from arrays, and its randomizing nature. Another unusual property is that the time needed to either add elements or to check whether an element is in the set is a constant, independent of the number of elements already in the set. Lastly, the k hash functions are independent and can be performed in parallel.

We propose to store legitimate IBPs into a Bloom filter and perform a validation by a membership query. We observe that unless the Bloom filter is not fully trained with a complete set of legitimate IBPs, it never gives any false alarms caused by the limited capacity of the physical structure of the IBP table. This is because we can always add more IBPs into the Bloom filter by setting more bits in the bit vector without discarding any existing information, thus effectively solving the overflow problem. This advantage, however, comes at the cost of incurring false positives. Notice that a false positive 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 carefully devise the filter parameters, i.e. m, k, n, and the hash functions, to minimize FPR within hardware budget while not hampering performance too much.

1) Security Analysis of the Bloom filter

To exploit the inherent 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 PC that the attacker can succeed to change (maybe through buffer overflow), and the other for the target address where the attacker can succeed to put or utilize malicious code. For one program, the number of the useful IBPs to the attacker that might lead to a successful exploitation normally is limited. Given a sufficiently low false positive rate, it is impractical for an attacker to try many IBPs in order to find one that is useful and happens to be able to skip the check through a bloom filter. In this work, we have empirically set a design goal of the FPR in the order of *1e-6*. Considering the fact that the attacker needs to match a desired PC value with such a false positive case, the success of an attack seems practically impossible.

2) Hash functions

For our hardware implementation, we choose a simple hash scheme, named simplehash, which does hashing by shuffling and xor-ing the bits in the 32-bit PC and the 32-bit target address as shown in Fig-4. The simulation shows under our configuration (detailed later), the FPR of this simplified hash scheme is within the same magnitude order as that of the ideal case.

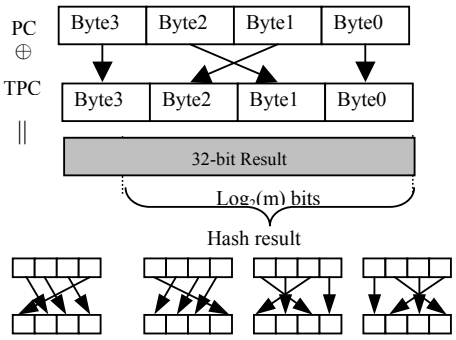

Fig-4. One of the Simplehash functions that shuffles PC and xors TPC to generate a hash result as well as other 4 hashing functions with different permutation.

Notice that unlike a software implementation, introducing more hashing functions/logic in hardware does not degrade the performance because hashing can always be performed in parallel. However, the bit vector must have more ports to support simultaneous reads/writes by more hash functions, which slows the access latency. In addition, hashing and bit vector accessing are performed independently therefore a

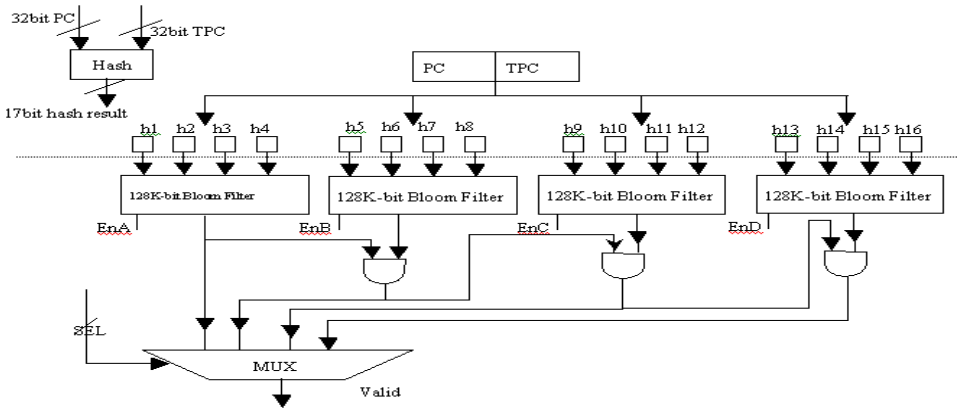

Fig-5. The validation unit cascading four 128K-bit Bloom filters as well as some glue logics. By properly setting enable and select signals, the hardware allows one IBP to pass one or multiple validations depending on the number of IBP a program contains and desired FPR.

bloom filter query can be naturally pipelined into 2 stages in a fast-clocked processor (see Fig-5).

3) Hardware organization

From Fig-1, we can see that *gcc* has the most IBPs (~10,000) in SPEC2000. Assume in the worst case an application has 16K IBPs (or n = 16K). Also assume a reasonable port number of a basic Bloom filer is four (k = 4), i.e. one Bloom filter is associated with four hash functions. According to *eq.* (2), FPR is minimized when m is 92K bits. So let us use 128Kbit as the size of a basic unit and *eq.* (1) gives a FPR of 0.024, which may be too high. Our design to reduce the FPR is cascading four 128K Bloom filters with *independent* hash functions shown in Fig-5. When enabling all four 128K-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. Table-2 presents the both FPRs of an ideal design that consists of a big multi-ported Bloom filter with perfect hash functions and of a cascading style design that includes several small basic Bloom filters with Simplehash functions. The former, which can achieve a 1.2 to 5.3 times lowerer FPR in theory, has to be heavily multi-ported. This is not a scalable solution in implementation because both the access speed and the power consumption of this memory-like structure (bit vector) increase dramatically as more ports are introduced. The latter has a higher FPR due to the less perfect hash functions and smaller size of bit vectors but still satisfies the design goal and is more scalable.

TABLE-2: THE FPRS FOR THE IDEAL CASE (ONE HEAVILY MULTI-PORTED BIG BLOOM FILTER WITH PERFECT HASH FUNCTIONS) AND SIMPLEHASH CASE (FOUR SMALL BLOOM FILTERS CASCADED USING SIMPLEHASH).

n	m (ideal/ Simplehash)	k	Overall FPR in 10^{-8} ideal	Simplehash
1000	128K / 128K×1	4/ 4×1	90	110
4000	256K / 128K×2	8/ 4×2	3.6	19
8000	384K / 128K×4	12/ 4×3	1.37	8.9
16000	512K / 128K×4	16/ 4×4	33	178

Another benefit of the cascading design is that a processor can adaptively enable only a subset of the Bloom filters depending on the IBPs a program contains and the false positive rate desired, thus reducing power consumption further. Fig-1 shows the IBP number varies significantly. For example, the programs that generate less than 1000 IBPs (e.g. *galgel, bzip2*) can use only one basic unit, i.e. EnA is asserted and the MUX selects input A. With m =128K, n=1000, and k=4, the ideal and simulated FPR is 9.0e-7 and 1.1e-6, respectively, which satisfies the design goal. The programs having less than 4000 IBPs (most cases in SPEC2000) will use two 128K-bit Bloom filters, i.e. both EnA and EnB are enabled and the MUX selects input B. Similarly, three or four basic Bloom filters are enabled for the program that contains more IBPs.

Generally, by using Bloom filters, we provide a mechanism that users can customize (i.e. changing the effective Bloom filter parameters) according to the security requirements, performance degradation tolerance, and hardware budget. Similar to encryption algorithms, the chance to break our protection is determined by the filter's parameters and mathematically predictable.

IV. ACCOMMODATING THE IBP TABLE INTO PIPELINE

A. Validating IBPs only on mis-prediction

We find that many processors are already doing a portion of validation in the form of indirect branch prediction. 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 zero and gradually filled up with targets that have been used at run-time. Since the validation unit 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 from 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 and the validation can be activated *only* on that event, rather than every instance of indirect branches. As a result, the existing prediction units effectively function as a "cache" for the IBP table.

Our simulation shows on average, for every 1000 committed instructions, there are about 1.02 mis-predictions for indirect branches in SPEC2000 benchmarks, resulting in a validation frequency of once every 1000 instructions. In contrast, a naive validation on every control flow transfer will occur roughly once every 10 instructions. This 100-fold reduction of the validation frequency makes our run-time control flow validation attractive in practice.

A worthwhile point is that we validate not just the target but also the IBP, i.e. the PC of the indirect branch as well as the target address. Indeed, a set-associative BTB found in many processors has a PC as the tag and compares it in prediction. The RAS only contains targets without PC information. However, processors must follow a restrictive way to read and update RAS because of the FILO feature of a stack. This fact suggests that it is possible, but very unlikely, for an attacker to find such a call and return sequence that he can corrupt the stack meanwhile still achieving an RAS hit, i.e. no mis-prediction.

B. *Fit into the pipeline*

As the validation against the IBP table (or the Bloom filter) is activated only on a mis-prediction, it is located in the branch verification stage when the computed target PC is available. Fig-6 illustrates how the Bloom filter is accommodated into the branch verification pipeline stage.

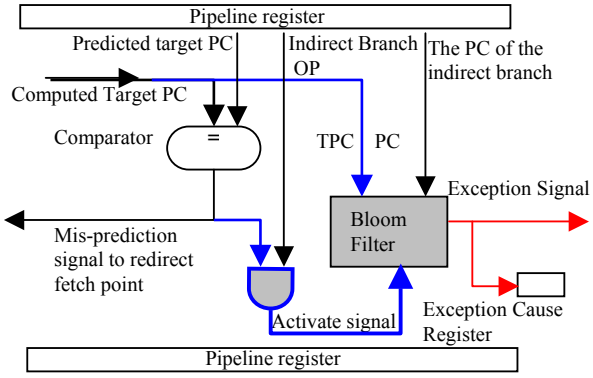

Fig-6. The Bloom filter is accommodated into the branch verification stage. The mis-prediction signal, generated by target address comparator, activates the Bloom filter.

On a branch mis-prediction, a mis-prediction signal is fed back to the fetch unit to redirect the fetch point. Meanwhile, the mis-prediction signal activates a validation or query in the Bloom filter for the encountered IBP generated by the program. If no such IBP is found in the Bloom filter, the Exception signal will be set to indicate that a suspicious branch is under execution. An exception reason may be written into the exception cause registers. The extra time to check the Bloom filter in our scheme appears as an added mis-penalty in the context of performance. As we shall see later, since the case of the mis-prediction is rare, the performance impact is nearly negligible.

V. Performance Evaluation

A. *Access Delay of the Validation Unit*

This section investigates the performance impact of our control flow validation mechanism using a cascaded Bloom filter with the Simplehash functions. The delay for accessing a 128K-bit vector is measured by a simulation based on CACTI 3.2[5] with .09µm technology. The storage structure is assumed to have four read ports and one to four write ports (Note that simultaneous writes are not necessary in training phase). Since CACTI can only simulate a minimum output size of 64 bits, we also add a 6-64 MUX in the data path. The delay of this MUX, as well as other components such as hashing logics and select logics, is estimated by a Verilog HDL implementation and a synthesis with TSMC's .09µm library. The simulation results are presented in Table-3.

Table-3: Access Delay of a Validation Unit

Simplehash logic	128K-bit vector with 1/4 write port(s).	6-64MUX Select logics	Total delay
0.49 ns	1.023 / 1.774 ns	0.79 ns	2.30/3.05 ns

Assuming a processor runs at a clock rate of 2.0 GHz, the access latency of the bit vector in the Bloom filter ranges from 5 cycles to 7 cycles. Considering possible circuitry-level optimizations and other extra overhead, we assign 4 to 8 cycles for accessing the validation unit to evaluate the performance impact.

B. *5.2 Performance Impact*

We tested SPEC2000 benchmarks running in Simplescalar that models an out-of-order 4-issue superscalar processor. No special design for indirect branch prediction is assumed and a conservative configuration of BTB and RAS is employed in the simulation (see Table-4). The reference input is used and the number of instructions specified by SimPoint[21] is skipped. To measure the performance impact, we impose extra delays (4 to 8 cycles as estimated earlier) for indirect branch mis-prediction penalty when the validation unit is incorporated.

Table-4: Architecture Parameters.

Parameter	Value
BTB	512 set, 4-way set associative
RAS	8 entries
Branch miss penalty	7 cycles
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	64K, 2 way set-asso., 2-cycle hit time
D-cache	64K, 4 way set-asso., 2-cycle hit time
L2 cache	Unified, 512KB, 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 FP Adder, 1 FP MUL/DIV

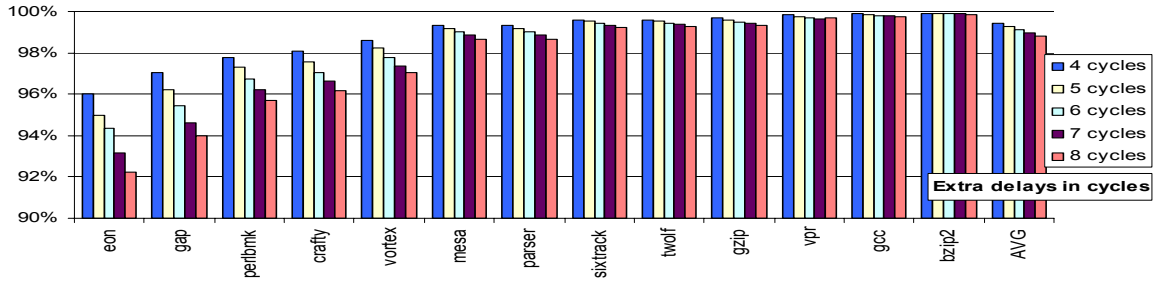

Fig-7. Normalized IPC (to the baseline case without any extra delays for validation). Only the benchmarks that have IPC degradation more than 0.1% are shown.

Fig-7 shows the resulting performance degradation with different validation unit access time. The increased mis-prediction penalty for indirect branches due to adding a Bloom filter has, on average, a negligible impact for most benchmarks. 12 out of 26 SPEC2000 benchmarks have a performance degradation that is less than 0.1%. Only six benchmarks, namely *eon*, *gap*, *perl*, *crafty*, *mesa*, and *vortex*, have an observable IPC decrease of more than 1%. The average IPC degradation is 1.2% while the worst performance drop is 7.7% for eon.

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! Our scheme also does not assume any special design for the indirect branch predictor. In literature, many works have been done about improving prediction accuracy for indirect branches[8]. Employing a more aggressive predictor can certainly reduce mis-predictions further, resulting in even less performance impact when our Bloom filter validation is employed.

VI. Related Works and Comparisons

Forrest[10] first characterizes normal program control flow transfer in terms of sequences of system calls. Wagner[24] (refined later by[11]) proposes to statically generate a non-deterministic finite automaton (NDFA) from the global control-flow graph of the program. Sekar[20] et al. proposed to generate a compact deterministic FSA coupled with PC value and system call to monitor the program execution at runtime. Because system call sequence itself has only a limited amount of information, it is subjected to mimicry attack and impossible path exploration. Note that none of the above validates control flow transfers directly due to huge run-time overheads. Our design explicitly validates control flow transfers invoked by indirect branches and monitors more precise execution information, i.e. indirect branch and its targets, in micro-architectural level. Thus, it is expected to be more efficient and have a shorter training time.

The earliest architecture proposal for control flow validation comes from the Data Mark Machine by Fenton[12], in which every memory word is enhanced with a tag. Based on the underlying security policy, the tag can be set and checked for potential security violation such as unauthorized "hidden" information flow. As a simple variation 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 channel[9][10]. With a sophisticated tracking and/or marking mechanism, if ultimately a datum marked with an un-trusted tag is used as control data, the system will capture the event and intercept the execution. Recently, the schemes that attempt to protect the control data directly were proposed by encrypting the control data value[15][23]. The control data value is hidden under encoding with a secret key until the last minute of its use for control flow change in program execution. An attacker, without the knowledge of the key, cannot read or write the control data correctly. Many other efforts[7] were also proposed to intercept some specific subset of control data attacks. However, our scheme can protect a much broader range of control data attacks, regardless of exploitations on a return address or a GOT entry or a general function pointer. It is also independent of the attack method in that it can intercept the attacks no matter whether it is through stack smashing, format string error, or directly pinpointing the control data and overwriting it.

Program shepherding[13] is an interpreter-based solution that collects legitimate branch targets when a runtime optimizer constructs traces. Any unauthorized branch target is detected at the loading of the traces. Besides the distinctions of their software vs. our hardware approaches, program shepherding uses a set of general capability rules to detect the attack while ours is more like a model-based solution. Program shepherding stores the validated execution path in the program address space. This, together with its sophisticated cache management algorithm, causes considerable memory overhead (3.02MByte vs 512Kbit hardware memory) and performance degradation (up to 170% vs 7.7%) comparing to our solution. Another similar work[25] also validates control flow transfer in hardware. But it mainly focuses on direct jumps and uses a sophisticated co-processor. This perhaps makes it less favored than our Bloom filter design in practice.

VII. Conclusions and Future Work

Current processor architecture is vulnerable for control flow altering attacks because control flow tracking is blindfolded without a validity check. This paper has proposed a practical solution to validate the control flow transfers via indirect branch instruction, which is the most likely target in control

data attacks. We suggest protecting both the location of the indirect branches and their target addresses (IBPs) by validating their legitimacy. All legitimate IBPs allowed under a given security policy will be collected and stored in a table first, and then the table is used to validate any control flow transfer at run-time. To our best knowledge, this work is the first effort that utilizes Bloom filter to represent the legitimate IBPs in the context of validating control flow transfer. A cascading-style Bloom filter design allows us to achieve a desired false positive rate at a small timing penalty. Moreover, our scheme safely activates the validation only on mis-prediction of the indirect branches, resulting in little performance degradation. A processor core with secure control flow creates a basis for designing and building more trusted devices and cyber infrastructure.

Having more context information beyond the IBP such as branch (path) history or the current frame pointer (FP) value will be useful to prevent more sophisticated attacks like the "impossible path" attack. The Bloom filter we have proposed eases this enhancement because the control data is hashed into a bit vector for its existence and we only need to adjust the Bloom filter configuration to keep the context information accurate. Exploring more context information beyond the IBP in full-scale web/Internet applications will be a part of our future work.

ACKNOWLEDGMENT

This work is supported by NSF Grants (ITR-0242222 and CT-0627341).

REFERENCES

[1] T. Austin and D. Burger, "The SimpleScalar Tool Set". *Univ. of Wisconsin CS Dept. Technical Report, No. 1342,* June 1997.

[2] B. Bloom, "Space/Time Tradeoffs in Hash Coding with Allowable Errors", In *Communications of the ACM* 13:7, 1970

[3] blexim, "Basic Integer Overflows", *Phrack, Volume 0x0b, Issue 0x3c, 2002*

[4] A. Broder, M. Mitzenmatcher, "Network applications of Bloom filters: A survey". *Annual Allerton Conference on Communication, Control, and Computing,* Oct. 2002.

[5] Cacti3.2. http://research.compaq.com/wrl/people/jouppi/CACTI.html.

[6] CERT Security Advisories. http://www.cert.org/advisories/

[7] C. Cowan, C. Pu, D Maier, J. Walphole, P Bakke, S. Beattie, A. Grier, P Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks", *In Proc. 7th USENIX Security Symposium,* Jan 1998.

[8] P. Chang, E Hao, Y. Patt, "Target prediction for indirect branches". *Proc. of the 24th ISCA*, 1997.

[9] J. Crandall and F. Chong, "Minos: Control Data Attack Prevention Orthogonal To Memory Model", *Proc. of the 37th Int'l Symp. on Microarchitecture,* Dec. 2004.

[10] S. Forrest, S. Hofmeyr, A. Somayajo and T Longstaff, "A Sense of Self for Unix Processes", *in Proc. of the 2000 IEEE Symp.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] J. Fenton, "Memoryless Subsystems". *Computer Journal, Vol. 17, no. 2, pp. 143-147* Feb. 1974.

[13] V. Kiriansky, D Bruening, S. Amarasinghe, "Secure Execution via Program Shepherding", In *Proc. of the 11th Usenix Security Symp*, 2002.

[14] C. Ko. C. Fink, K. Levitt, "Automated detection of vulnerabilities in privileged program s by execution monitoring", *in Proc. of the 10th Computer Security Applications Conference,* 1994.

[15] G. Lee and A. Tyagi, "Encoded Program Counter: Self-Protection from Buffer Overflow Attacks", *Proc. of the First Int'l Conference on Internet Computing,* June, 2000

[16] P.S. Magnusson, M. Christensson, J. Ekilson, D. Forsgren,, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, B. Werner, "Simics: a full system simulation platform", *IEEE Computer* (February), 2002

[17] Anonymous, "Once upon a free()". *Phrack, 9(57),* Aug., 2001

[18] A. One. "Smashing the stack for fun and profit". *Phrack, 7(49),* Nov. 1996.

[19] scut / team teso, "Format Exploiting Format String Vulnerabilities", Sept, 2001.

[20] R. Sekar, M. Bendre, P. Bollineni, D. Dhurjati, "A fast Automaton-Based Method for Detecting Anomalous Program Behaviors", *in Proc. of the IEEE Symp.on Security and Privacy*, 2001.

[21] T. Sherwood, E. Perelman, G. Hamerly and B. Calder, "Automatically Characterizing Large Scale Program Behavior", In *Proc. of the 10th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems,* Oct. 2002.

[22] G. Suh, J. Lee, S Devadas, D. Zhang, "Secure program execution via dynamic information flow tracking". In *Proc. of the 12th Int'l Conf. on Architectural Support for Programming Languages and Operating Systems,* 2004.

[23] N. Tuck, B. Calder, G. Varghese, "Hardware and Binary Modification Support for Code Pointer Protection from Buffer Overflow", *Proc. of the 37th Int'l Symp. on Microarchitecture,* 2004.

[24] D. Wagner and D. Dean, "intrusion detection via Static Analysis", *in Proc. of the IEEE Symp.on Security and Privacy*, 2001.

[25] T. Zhang, X. Zhuang, W. Lee, S. Pande, "Anomalous Path Detection with Hardware Support," *in Proc. of CASES*, 2005.