An Efficient Hardware Support for Control Data Validation

Yong-Joon Park, Zhao Zhang
Department of Electrical
and Computer Engineering
Iowa State University
Ames, IA, 50011, USA
{ypark,zzhang}@iastate.edu

Gyungho Lee
Department of Electrical
and Computer Engineering
University of Illinois at Chicago
Chicago, IL 60607, USA
ghlee@uic.edu

Abstract

Software-based, fine-grain control flow integrity (CFI) validation technique has been proposed to enforce control flow integrity of program execution. By validating every indirect branch instruction, it can prevent various control flow attacks, but at the cost of non-trivial overhead: up to 50% and on average 21% as reported in a case study. We propose a new hardware mechanism to accelerate the CFI validation. It utilizes the branch prediction unit of modern processors to reduce the frequency of necessary validation, and proposes to use a small hardware structure called indirect branch filter cache (IBF cache) *to further reduce the frequency of validation. The small IBF cache buffers and reuses previous validation results, which dramatically reduces the frequency of validation for all workloads we have studied. We collect the trace of indirect branch of various workloads on an Intel P4 computer and conduct trace-based simulation to estimate the performance overhead. Our results show that the overhead is negligible for all SPEC CPU2000int, SPEC CPU2006int programs, TPC-C, WebStone and FTP server benchmarks.*

1 Introduction

Control flow attack have been frequently exploited to make the first breach to computer system security. Many service programs have unintended vulnerabilities in their dynamic execution environments, which are explored by attackers to redirect their control flow to harmful code. Control flow attacks not only cause breaches to individual systems; more often, they lead to large-scale attack such as distributed DOS (denial of service) attacks and may result in monetary loss of millions of dollars. Unfortunately, the vulnerabilities leading to control flow attacks are inherent and pervasive in software as reported by the National Vulnerability Database (NVD) [11].

To prevent those attacks, one of the solutions can be validating all indirect branches as their *branch and target* address pair. However, it is also necessary to implement this type of validation with hardware support for the overhead to be reasonable [1, 2, 15, 13]. Abadi et al. [1] proposed control flow integrity (CFI) enforcing mechanism by using binary instrumentation techniques. It identifies valid target and breach pair statically and inserts checking routine before a indirect branch instruction. It demonstrates the effectiveness of the mechanism by testing various attack scenarios. However, it incurs non-negligible performance overhead: up to 50% and on average 21% for SPEC CPU2000int benchmarks. In this study, we propose an efficient design to validate every *indirect branch* as an address pair of *(source, target)* with pre-generated valid set of address pairs, which can be generated statically by compilers or dynamically by run-time systems.

The core part of the design is a small *IBF Cache* (Indirect Branch Filter Cache), which dramatically reduces the performance overhead at the cost of a small hardware cache. With IBF cache, the performance overhead is negligible for all workloads we have studied. The IBF cache buffers and reuses recently validated address pairs. Its structure is somewhat similar to that of a BTB (branch target buffer); however, it uses an XOR-based indexing schemes to eliminate the conflicts from multi-target indirect branches, which are common to the BTB structure[1]. Our simulation results show that the miss rate of the IBF cache (2K entries by default) is less than five misses per 100,000 indirect branches on average for the SPEC CPU2000int benchmarks; and less than one miss per one million committed instructions. The good performance is the result of good temporal locality in program control flow. Since the overall performance overhead is a product of the miss frequency and the per-validation overhead, the IBF cache reduces the overall performance overhead significantly. Furthermore, it is

[1]The IBF cache is accessed at the execution stage when the branch target address becomes known, so it can use both the branch and target addresses in its indexing.

now possible to use relatively complex validation methods, which may use off-chip storage.

We have evaluated the hardware design using a wide range of workloads, including the SPEC CPU2000int and CPU2006int programs, TPC-C workloads, WebStone benchmarks, and an FTP server benchmark. The performance overhead is virtually non-existent for many workloads and less than 2% for all benchmarks.

The rest of the paper is organized as follows. Section 2 introduces the background and the related work. Section 3 describes our design of the IBF cache and control flow validation. The experiment environment is described in Section 4. Section 5 presents the experiment results and analysis. Finally, Section 6 discusses limitations and future work.

2 Background and Related Work

Background Control flow attack is an elementary form of attack to computer systems, which is commonly used as the first step to gain the access to a system. It exploits the vulnerabilities in software, such as the lack of boundary checking, to alter the normal program control flow. That is usually done by overwriting *control data*, which is the address of the next instruction to be executed such as function return address, function points, or jump table address. One of the well known example is stack-based buffer overflow attack; the normal control flow is altered by rewriting the return address stored in the program stack. Those attacks may be prevented by compiler or hardware defense mechanisms [4, 6, 12]. However, control flow attacks may take sophisticated forms that are more difficult to prevent than the stack-based buffer overflow attack. For example, it may overflow control data in data or heap segments, such as function pointers, GOT, *.dtors* and virtual function table. It may also make the control flow jump to existing code such as library functions making system calls, which are safe with normal control flow but may cause damages in abnormal execution. Unfortunately, the related vulnerabilities are widely spread in may programs.

Currently, most microprocessors have branch prediction facilities; branch target buffer (BTB), branch history table (BHT) and return address stack; and some processors have enhanced support for indirect branch prediction. BTB improves performance by predicting the target of branch by using stored branches target recently taken. Hence it may reduce the frequency of the control flow validation by checking only when branch predictor predicts the target incorrectly for indirect branch. Note that all the branches stored in the branch predictor have been validated before due to cold miss. Consider a simple example: A program has a simple loop that calls a function, which performs simple calculation, via indirect function calls over million times. A software based approach will do one validation for each function call or return, while in our approach only two validations are needed for the whole loop.

Software/Hardware Methods to Maintain Control Flow Integrity One type of work is to validate the execution of every branch instruction with the valid control flow information of the program. CFI [1] is software fine-grain control flow integrity checking mechanism. It instruments the program executable to insert a label right before each function or a code block. The label is write protected because it is in the code segment. Every indirect branch[2] instruction is also rewritten as a small piece of code that checks the label before the jump. The labels are constructed from static analysis of the control flow graph (CFG) of the program. When compared with our work, the overall performance overhead is significant. Additionally, in certain cases, the use of label cannot preserve the precise CFG information; for example, if two indirect branches share one target (and the associated label) but not other targets. The overhead is non-trivial: up to 50% and on average 21%. A hardware version of CFI with ISA extension [2] was later implemented to reduce the overhead. New instructions are introduced to replace the guard code with a single instruction. In the initial performance evaluation, five integer benchmarks from SPEC CPU2000int are observed with test input set. They reported that the results showed maximum 7% performance overhead and around 2% on average. Nevertheless, the aforementioned issue of label sharing still exists.

Zhang et al. [15] also propose a hardware mechanism to validate branch address pairs and control flow path with dynamically generated control flow information. Their hardware design, however, is more complex than ours because of the use of hash table and other complex structures to store the control flow information. In our design, the IBF-cache makes the whole hardware design very simple. The reported performance overhead is about a few percent in the study, but the evaluations are done with much shorter relative memory latency which 30 processor cycle since it is targeting embedded system.

Hardware Information Tracking Another form of hardware protection can cover attacks to all memory regions but requires much more extensive changes of the CPU internals. Suh et al. [14] proposed dynamic flow information tracking that tracks every datum in the memory to see whether the original source is from external I/O, which is suspicious, or from the program internal, which is trustworthy. As for the hardware changes, an one-bit tag is required for every memory datum and CPU registers, and CPU data paths are changed to propagate the tags during address calculation. When suspicious data is used as control data, it raises an alarm. Crandall and Chong [5] proposed *Minos* that also tracks the information flow in program execution using one-bit tags. Instead of tagging spurious data rooted from I/O,

[2]This paper uses the generic term "indirect branch" and does not distinguish conditional indirect branch and unconditional indirect jump.

	Total # Indir	Total # Pairs	90% Indir	90% Pairs	99% Indir	99% Pairs
gzip	157	274	4	28	8	42
gcc	1921	10099	300	4435	733	7234
mcf	178	289	4	9	16	37
crafty	310	1410	25	527	42	616
parser	455	1327	37	326	95	590
eon	875	2426	53	331	85	365
perlbmk	427	1164	22	56	27	61
gap	1014	4233	134	2006	379	3026
vortex	739	3697	46	883	162	1695
bzip2	146	293	6	50	9	86
twolf	325	1264	37	213	72	671

Table 1. Indirect branch profiling for SPEC CPU2000int.

Minos uses the bit to mark high-integrity data, which can be used as control data. An alarm will be raised if a low-integrity data is used as a target address. In the Minos implementation, data created before a timestamp is marked as high integrity. Additionally, the values rooted from the program counter are also of high integrity.

Run-time Systems Program shepherding [9] is derived from DynamoRIO runtime optimizer and detects unauthorized control transfer. It translates executable binaries and generates code blocks. In order to achieve low performance overhead, DynamoRIO uses software code cache to execute newly constructed fragments natively. When the control transfer occurs, DynamoRIO code gets the program control and dispatches the instructions from code cache. Since the address of instructions in code cache differs from the address in original binaries, it requires address lookup in control flow transfer cases. Therefore this mechanism can identify the control instructions and verify the control data. It establishes general security policy to restrict certain types of control transfers and activities.

3 Hardware Support for CFI Monitoring

The proposed CFI validation system consists of the hardware IBF-cache and software CFI monitoring tool. In a proof-of-concept design, we have used the performance monitoring [8] features of the Intel Pentium 4 processors to validate the effectiveness and analyze the performance overhead. We configure the performance monitoring features to generate an interrupt on every indirect branch misprediction. The software CFI monitoring tool obtains the control and verifies the CFI upon the interrupt. In other words, the tool implements software-based validation and utilizes the branch prediction unit of the Pentium 4 processor to reduce the frequency of validation. We have had two findings: First, the software tool can successfully intercept control flow attacks for various workloads we tested, including SPEC programs, web server and other programs. Second, the performance overhead is still comparable to that of the previous study [1]. Our analysis shows that the overhead comes from the high interrupt and validation latency. However, if the frequency of validation can be further reduced, the software validation approach can be very efficient.

Intel Pentium 4 processor [8] uses branch history table, branch target buffer and return address stack for indirect branch prediction. Although the processor achieves low branch mis-prediction rate, software CFI monitoring tool still incurs non-negligible performance overhead. Therefore, we propose the use of IBF-cache to reduce the frequency of the validation. The IBF cache records the indirect branch address pairs that have been validated recently. It is a small component and is as fast as L1 caches. For a mispredicted indirect branch, the validation starts at the execute/writeback stage where the branch and target addresses are known. This address pair is sent to the IBF cache. If it hits in the cache, then it has been validated. This validation can be done within branch mis-prediction penalty cycle, hence there will be no overhead for accessing IBF-cache. Upon IBF cache miss, it generates an interrupt for CFI monitoring tool to valid control flow with the information stored in the off-chip main memory for expensive validation. If the outcome is negative (no alarm), the monitoring tool returns from the interrupt routine. Otherwise, it raises exception and let software further examines the program.

The IBF cache uses an XOR-based indexing scheme, which are from both the branch address and the target address, to avoid misses from indirect branches which have multiple targets during the program execution. If an indirect branch uses multiple targets, the XORing of the target address will distribute all branch address pairs of this branch over the cache address space. Otherwise, those branch address pairs will be mapped onto the same cache entry, causing severe conflicts. XOR-based indexing scheme has been used in both cache indexing [7] and branch target address prediction (XORing branch address with branch history). Nevertheless, the branch target address prediction cannot use the XORing with branch target address because it is unknown at the time of branch prediction. This is the reason the IBF cache may not be replaced by an enlarged BTB.

4 Experimental Methodology

We collected the trace of indirect branch misprediction on Intel Pentium 4 processor using the Intel VTune Analyzer. The trace is then fed into a trace-based simulation of the IBF cache. We also measured the interrupt and validation overhead. This is done by using our proof-of-concept software monitoring tool to count the number of validations; and by measuring the difference of program running times with and without enabling the software monitoring tool. We then estimate the overall performance overhead from the per-validation overhead and the frequency of validation.

We have used various workloads in our performance evaluation: TPC-C workload with *Postgres* 7.4.13 database system, *WebStone 2.5* benchmark with *Apache 2.0.47*, and *dkftpd* benchmark, FTP benchmark, with *vsftpd* demon. We also evaluate *SPEC CPU2000int* and *SPEC CPU2006int* benchmarks. All programs are compiled with gcc 3.3.2 and run on Redhat Linux with kernel 2.4.26.

- SPEC CPU2000int and SPEC CPU2006int benchmarks: We use the reference input sets and run all programs to completion. The CPU2006int programs have generally more complex source code than the CPU2000int programs.
- TPC-C workload: TPC-C is an OLTP (on-line transaction processing) workload that emulates warehouse transactions using a database system. We use Postgres 7.4.13 as the supporting database system.
- WebStone 2.5 benchmark: The WebStone benchmark creates a load on a Web Server by simulating the activities of multiple clients. We configure WebStone to use two different types of access methods, HTML and CGI, with 10 to 100 simultaneous clients. The underlying web server is Apache 2.0.47.
- FTP Workload: We use an FTP demon called *vsftpd*, version 1.2.0-5, and an FTP benchmark called *dkftpbench*.

5 Experiment Results

5.1 Profiling of Indirect Branches

We first collected the indirect branch profiling on the Intel Pentium 4 processor. We found that the indirect branch makes up about 1.5% of all instructions. The mis-prediction ratio for the SPEC integer benchmarks is no more than 10% and on average 3.1%. Table 1 shows the indirect branch profiling for SPEC CPU2000int benchmarks. We do not include CPU2000fp benchmarks because those floating-point programs are much less branch-intensive and therefore the performance overhead for them is not a concern. The table shows that a small subset of static indirect branches make up a large portion of dynamic indirect branches. The second and third columns of the table are the total number of static indirect branches and the total number of unique targets, respectively. The fourth column is the number of static indirect branch instructions that are responsible for 90% dynamic indirect branch instructions. The fifth column is the number of unique indirect branch address pairs that are observed in the execution of 90% indirect branch instructions. The sixth to ninth columns are similar except that ratios 95% and 99% are used. The profiling results show that it is very promising to use a small cache to capture the locality existing in the indirect branch address pairs observed in the program execution; and the frequency of validation may be significantly reduced by the IBF cache.

5.2 Results for SPEC Benchmarks

Table 2 shows the IBF cache miss rates for all SPEC CPU2000int and CPU2006int programs with the reference input sets. The CPU2006int programs are of more interests because they have more complex source code. Additionally, there are three more C++ programs, omnetpp, astar, and xalancbmk, which may have high frequency of indirect branch instructions. For both benchmarks, the IBF cache miss rate becomes very small when the cache size increases beyond 1K; the maximum for CPU2006int is 0.898% on *gobmk* for the cache size of 1K-entry. The average miss rate of CPU2006int with 8K-entry is 0.128%. The three new C++ programs are not very different from the other programs. Comparing CPU2006int with CPU2000int, the average miss rate increases slightly for all cache sizes.

Table 3 shows the number of IBF cache misses per 10,000 instructions for SPEC CPU2000int and SPEC CPU2006int programs. This number is closely related to the overall performance overhead. It is determined by three factors of a program: the ratio of indirect branch instructions, the branch mis-prediction rate and the IBF cache miss rate. Program *gcc* has the largest number for all cache sizes mainly because it has relatively high frequency of indirect branch instructions. We can give a ballpark estimate of the overall performance overhead for CPU2000int *gcc* with an IBF cache of 2k entries as follows: Assume that the off-chip validation averagely take 469ns or 1500 cycles on a 3.2 GHz processor, which is measured using the aforementioned software monitoring tool we developed; and assume the program CPI is 1.81 [3]. The performance overhead is about $1500 * 0.091/(10,000 * 1.81) = 0.65\%$. Similarly, we obtain an average performance overhead of 0.0059% for CPU2000int programs (Miss rate is one miss per million instructions and CPI is 2.545 on average). In fact, this estimate is pessimistic because Intel processors has relatively high latency in accessing the performance monitoring counters. Even so, the performance is very good when compared with the previous work [1, 2], which has up to 50% overhead and on average 21% overhead. The results for other IBF cache configurations are not included because of the space limit. We found that a two-way set associative IBF cache improves significantly over the direct mapped one with the same size; and that the improvement diminishes when the degree of associativity increases beyond four.

5.3 Results for Other Workloads

The other workloads include TPC-C, WebStone and an FTP server benchmark; see Section 4 for their description. Table 4 shows the IBF cache miss rate with varying cache size from 1K to 8K with the set associativity fixed at four. The TPC-C workload is known to have large instruction footprint, and therefore the result indicates that the control flow in TPC-C has good locality. The WebStone benchmark

Program	1k	2k	4k	8k
gzip	0.002%	0.002%	0.002%	0.002%
vpr	0.002%	0.002%	0.002%	0.002%
gcc	0.415%	0.297%	0.202%	0.161%
mcf	0.591%	0.591%	0.591%	0.591%
crafty	0.172%	0.110%	0.106%	0.102%
parser	0.271%	0.085%	0.030%	0.028%
eon	0.001%	0.001%	0.001%	0.001%
perlbmk	0.718%	0.046%	0.024%	0.001%
gap	0.042%	0.016%	0.003%	0.001%
vortex	0.015%	0.012%	0.012%	0.012%
bzip2	0.003%	0.003%	0.003%	0.003%
twolf	0.002%	0.002%	0.002%	0.002%
Average	0.186%	0.097%	0.081%	0.075%

(a)

Program	1k	2k	4k	8k
perlbmk	0.071%	0.012%	0.005%	0.004%
bzip2	0.009%	0.009%	0.009%	0.009%
gcc	0.148%	0.089%	0.055%	0.030%
mcf	0.020%	0.020%	0.020%	0.020%
gobmk	0.898%	0.351%	0.117%	0.055%
hmmer	0.023%	0.022%	0.022%	0.022%
sjeng	0.006%	0.004%	0.003%	0.003%
libquantum	0.695%	0.695%	0.695%	0.695%
omnetpp	0.007%	0.004%	0.003%	0.003%
astar	0.568%	0.563%	0.562%	0.562%
xalancbmk	0.320%	0.053%	0.007%	0.004%
Average	0.251%	0.166%	0.136%	0.128%

(b)

Table 2. The IBF cache miss rates for SPEC (a) CPU2000int and (b) CPU2006int benchmarks with the reference inputs for cache sizes of 1k to 8K entries. The cache set associativity is fixed at four. The numbers in bold type are the maximum number for a given cache size.

Program	1k	2k	4k	8k
gzip	0.000	0.000	0.000	0.000
vpr	0.000	0.000	0.000	0.000
gcc	0.131	0.091	0.060	0.046
mcf	0.000	0.000	0.000	0.000
crafty	0.023	0.015	0.014	0.014
parser	0.004	0.001	0.000	0.000
eon	0.000	0.000	0.000	0.000
perlbmk	0.003	0.003	0.003	0.003
gap	0.021	0.008	0.001	0.000
vortex	0.000	0.000	0.000	0.000
bzip2	0.000	0.000	0.000	0.000
twolf	0.000	0.000	0.000	0.000
Average	0.015	0.010	0.007	0.005

(a)

Program	1k	2k	4k	8k
perlbmk	0.044	0.008	0.003	0.002
bzip2	0.000	0.000	0.000	0.000
gcc	0.313	0.147	0.088	0.058
mcf	0.000	0.000	0.000	0.000
gobmk	0.007	0.003	0.001	0.000
hmmer	0.000	0.000	0.000	0.000
sjeng	0.002	0.001	0.001	0.001
libquantum	0.000	0.000	0.000	0.000
omnetpp	0.001	0.001	0.000	0.000
astar	0.000	0.000	0.000	0.000
xalancbmk	0.017	0.003	0.000	0.000
Average	0.035	0.015	0.009	0.006

(b)

Table 3. (a) The number of IBF cache misses per 10,000 instructions for SPEC CPU2000int benchmark programs; and (b) that for SPEC CPU 2006int programs.

has very high miss rates with IBF cache of 1K entries or less, and drops to less than 1% when the size increases beyond 2K. The FTP server benchmark incurs few IBF cache misses for all sizes. The number is negligible for all the workloads for cache size of 4K or more entries.

Table 5 shows misses per 10,000 instructions for other workloads. The Web server benchmark shows high number of misses due to higher indirect branch ratio (4.17%) and 5.3% indirect branch mis-prediction ratio. Although IBF-cache performances of the benchmarks is worse than SPEC benchmarks, the performance degradation will not be noticed, since these programs spend most of time on I/O operations.

6 Discussion and Future Work

Implementation The proposed IBF cache will be a very small component in modern processors. Because it is accessed only on indirect branch mis-predictions, it can be away from the core pipeline logic on the processor chip and therefore will not complicate the chip layout design. The IBF design can be ported to virtually any pipelined, high performance processors a branch prediction unit. It may require the hardware support for raising an exception on IBF-cache miss, but does not require any change to the user-level ISA.

Storage Overhead The third column of Table 1 shows the total number of branch pairs in the SPEC CPU2000int benchmark. The largest number of the branch pair is 10,099 for gcc. Since the branch pair is stored in plain hash table in our design the storage overhead would be 16 bytes of branch and target address 2 bytes of link field and 2 bytes of anchor per each entry. Therefore the maximum storage overhead for the CPU2000int programs is around 198KB in the off-chip memory which can be negligible in current systems. We also find that other benchmarks also have smaller number of branch pairs (HTTPD 9570, TPCC 4346, and VSFTPD 2195). The IBF cache has branch address and target address per each entry. The IBF cache itself will consume 16KB for 2K-entry 32-bit addressing mode and 32KB

Workload	1k	2k	4k	8k
TPC-C	0.431%	0.081%	0.001%	0.000%
WebStone	24.496%	3.389%	0.463%	0.180%
WebStone CGI	29.596%	6.369%	0.643%	0.194%
FTP Server	0.125%	0.117%	0.116%	0.116%
Average	13.662%	2.489%	0.306%	0.123%

Table 4. The 4-way set associative IBF cache miss rates for other workloads with cache size of 1K to 8K entries.

Workload	1k	2k	4k	8k
TPC-C	0.071	0.013	0.000	0.000
WebStone	5.467	0.756	0.103	0.040
WebStone CGI	6.648	1.431	0.144	0.044
FTP Server	0.010	0.009	0.009	0.009
Average	3.049	0.552	0.064	0.023

Table 5. IBF cache cache misses per 10,000 instructions for TPC-C, WebStone and FTP server benchmarks.

for 2k-entry 64-bit addressing mode.

Impossible Path Attack In this paper, we mainly focus on validating control data to prevent prevalent types of control flow attacks. Although the real attack has not been reported yet, impossible path attack [15] would bypass our detection mechanism. Impossible path attacks alter only control decision making data or both control decision making data and control data in the program. In order to detect the attack, Zhang et al [15] check the last n branch history information for every branches. They also implement a compiler to identify the impossible path. The impossible path attack detection mechanism can be easily adopted in our design: Intel Pentium 4 processor uses a branch history table and BTB to achieve prediction accuracy for all types of branch except return instruction, i.e. the branch predictor is already capable of checking last 16 branch history. Additionally, the processor has Last Branch Recording (LBR) stack to provide branch and target addresses of last 16 branches. With those two existing hardware facilities and the extension of IBF cache, we can extend our system to check the impossible path attack. We are currently extending our system to check the attacks by using above extension and adopting SRAS [10, 12] to further reduce the validation frequency.

7 Conclusion

We have proposed a highly efficient hardware mechanism to accelerate the fine-grain control flow validation. It uses IBF cache to minimize the performance overhead associated with frequent validation. We have evaluated the design using a wide range of applications, including SPEC CPU2000int, SPEC CPU2006int, TPC-C workloads, WebStone benchmark and an FTP benchmark, and found that the performance overhead is negligible.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity:principles, implementation and application. In *12Th ACM Symposium on Computer and Communication Security*, Alexandria,VA, 2005.

[2] M. Budiu, Ú. Erlingsson, and M. Abadi. Architectural support for software-based protection. In *Workshop on Architectural and System Support for Improving Software Dependability (ASID)*, San Jose, CA, 2006.

[3] M. Christoper Martinez and E. B. John. Multimedia workloads versus spec cpu 2000. In *Spec Benchmarking Workshop*, 2006.

[4] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, P. A. Grier, Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th USENIX Security Symposium*, page 63?8, San Antonio, TX, 1998.

[5] J. R. Crandall and F. T. Chong. Minos: Control data attack prevention orthogonal to memory model. In *Proceedings of the 37th Annual International Symposium on Microarchitecture*, Portland, OR, USA, 2004.

[6] M. Frantzen and M. Shuey. Stackghost: Hardware facilitated stack protection. In *In Proceedings of the 10th USENIX Security Symposium*, Washington, DC, 2001.

[7] A. Gonzalez, M. Valero, N. Topham, and J. M. Parcerisa. Eliminating cache conflict misses through XOR-based placement functions. In *Proceedings of 11th International Conference on Supercomputing*, pages 76–83, Vienna, Austria, 1997.

[8] Intel Corp. IA-32 Intel Architectures Software Developer's Manual. January, 2006.

[9] V. Kiriansky, D. Bruening, and S. Amarasinghe. Secure execution via program shepherding. In *In Proceedings of 11th USENIX Security Security Symposium*, San Francisco, CA, 2002.

[10] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi. Enlisting hardware architecture to thwart malicious code injection. In *Proceedings of the 2003 International Conference on Security in Pervasive Computing*, pages 237–252, Boppard, Germany, 2003.

[11] NVD. National vulnerability database, a comprehensive cyber vulnerability resource. `http://nvd.nist.gov`.

[12] Y.-J. Park, Z. Zhang, and G. Lee. Microarchitectural protection against stack-based buffer overflow attacks. *IEEE Micro*, 26(4):62–71, July/August 2006.

[13] Y. Shi, S. Dempsey, and G. Lee. Architectural support for run-time validation of control flow transfer. In *International Conference on Computer Design*, San Jose, CA, 2006.

[14] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. In *In Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, 2004.

[15] T. Zhang, X. Zhuang, S. pande, and W. Lee. Anomalous path detection with hardware support. In *Proceedings of International Conference on Compilers, Architectures and Synthiesis for Embedded Processors*, San Francisco, CA, 2005.