

Embedding Program Code Integrity Monitoring in Application-specific Instruction Set Processors *

Yunsi Fei

Dept. of Electrical & Computer Engineering
University of Connecticut
Storrs, CT 06269
E-mail: yfei@engr.uconn.edu

Z. Jerry Shi

Dept. of Computer Science & Engineering
University of Connecticut
Storrs, CT 06269
Email: zshi@engr.uconn.edu

Abstract

Program code in a computer system can be altered either by malicious security attacks or by various faults in microprocessors. In this work, we present a generalized methodology for monitoring code integrity at run-time in application-specific instruction set processors (ASIPs). We embed monitoring mechanisms in an ASIP design process. Thus the processor is augmented with a hardware monitor automatically. Experimental results show that our microarchitectural support can detect program code integrity compromises with small area overhead and little performance degradation.

1 Introduction

Recent years have seen two trends driving reliability and security to become critical concerns for embedded processors. As technological trends continue to lead towards smaller and faster transistors with lower threshold voltages and tighter noise margins, the probability of transient faults, also known as *soft errors*, has increased dramatically [6]. The other trend in embedded systems has been the drastic increase of embedded software contents and the pervasiveness of networked connections. The vulnerability of systems to software attacks has thus been increased. Security has emerged as a new system design goal in addition to the traditional design metrics of performance and power consumption [4]. In this paper, we focus on detection of program code compromises, no matter whether they are caused by soft errors or security attacks.

Recent decades have also seen the emerging of application-specific instruction set processors (ASIPs) as an important design choice for embedded systems. Since ASIPs allow both the ISA and underlying microarchitecture to be tuned for specific applications, they provide a good platform for integrating code integrity monitoring mechanisms into the design process. In this work, we propose a microarchitectural support for program code integrity monitoring in ASIPs.

Typical approaches to counter soft errors rely on redundant resources. A critical operation can be performed in several identical circuits simultaneously or in one circuit at different times. Any discrepancies among the results indicate soft errors. However, redundancy of hardware is normally expensive. SWIFT addresses the problem with software approaches [5]. The limitation of this method is that it does not detect multiple-bit faults and assumes that a processor has sufficient resources (registers and functional units) to execute redundant codes without significant performance degradation.

To prevent security attacks that execute malicious code, checkpoints can be placed at one or multiple layers in a system. However, if checking is done too early, it does not catch

any faults taking place at a later time. Our method checks the integrity of the code in the instruction fetch (IF) and decode (ID) stages.

We address the problem of program code integrity monitoring by ensuring that run-time program execution does not deviate from the expected behavior. A monitor, placed in the IF and ID stages in a processor, captures properties of the permissible behavior and compares the dynamic execution with it. When a mismatch is detected, the monitor throws an exception to trigger appropriate remedy mechanisms.

2 Design Rationale

Since hash values are a good indicator of program behaviors, we monitor program code integrity by comparing two hash values of the instruction streams. One is generated before the program starts and the other generated by the processor at run-time after instructions have been fetched into the processor. Although the design idea is drawn from standard code integrity monitoring approaches, many issues remain to be investigated to make code checking efficient and effective in a processor.

An appropriate granularity level to characterize the program's properties will affect the design complexity and effectiveness greatly. There are several considerations in deciding the number of hash values we need to compute and the range of instructions each hash value monitors. 1) The expected hash should be computed statically before the execution and will match the dynamic hash if the program is not compromised. 2) A behavior violation can be detected promptly, ideally, before any damages are inflicted. 3) The hardware and performance overhead involved in run-time checking should be reasonable. Considering the above requirements, we choose to monitor program code at the basic block level.

Another issue is where the code monitoring mechanism is located. We would like to place it in late stages, e.g., as close to the decode stage as possible, to catch more possible code changes. We decide to incorporate the monitoring mechanism into pipelines and perform the checking in the IF and ID stages. Any alterations that are made before instructions are fetched into processor pipeline will be detected.

In order to compute and compare hash values, the microarchitecture needs to be enhanced. An internal hash table (IHT) (or one or more special registers) is added to store expected hash values. At run-time, the enhanced hardware detects the beginning of a basic block and starts computing its hash value as instructions are being fetched. When the program proceeds to the end of the basic block, the IHT is searched. If a hash table entry for the basic block is found and the dynamic hash matches the expected one, it is a *hash hit* and the basic block is considered intact. If the basic block is found in the hash table but the dynamic hash does not match the expected one (defined as *hash mismatch*), or the basic block is not found in the hash table at all (defined as

*This work was supported by an NSF grant CCF-0541102.

hash miss), an exception is raised, with different signals indicating the cause. The OS will take over the control and decide how to respond.

The expected hash values can be loaded into the IHT by applications, as seen in [3], or the OS. If applications load the hash table, compilers need to insert at proper locations of programs the instructions that load expected hash values. The hash table load instructions will increase code size dramatically, and this method also increases complexity of the compiler.

Alternatively, the IHT can be managed by the OS. The compiler still needs to generate the expected hashes for each block. However, in this method, all the hash values are simply attached to the application code and data and will be loaded into a section of memory managed by the OS when the application starts. The hash values can even be computed after binary code is generated, e.g., by a special program or the OS application loader. At the end of a basic block execution, various monitoring results can be generated. A *hash mismatch* will signal the OS to terminate the application. On an exception caused by a *hash miss*, the full hash table (FHT) in memory will be searched instead, and some entries in the IHT will be replaced. If the basic block is not in the FHT either, or dynamic hash is different from the expected hash value, the OS will terminate the program. Note that the search of the FHT can be done either by hardware or by software, in a similar manner as a cache miss handler.

In this paper, we start with a widely adopted assumption of simple faults: considering only a single bit flip in a basic block of program code. We employ a simple checksum function, XOR, in our experiments. The probability of errors not being detected is small. As long as the number of bits changed is odd, the XOR checksum can always detect the error. In the future, we will improve the XOR scheme with process-dependent random values. We will also look into more secure yet efficient hash algorithms.

3 Experimental Results

We incorporated the code integrity checker into an automatic processor synthesis tool, ASIP Meister [1]. The checking mechanism is specified by microoperations and is embedded into the pipeline stages. The tool captures target processors' specification through a GUI and generates RTL processor descriptions for logic synthesis. Since microoperations are at a lower software architecture level than processor instructions, the microarchitectural support for program code integrity monitoring is transparent to upper software levels and no recompilation or modification is needed for the program. After the customized processor enhanced with code integrity monitoring is generated, we use ModelSim to simulate the VHDL code [2]. The functionality of applications is verified as well.

The custom processors were synthesized into technology-mapped gate-level netlists using Synopsys Design Compiler with TSMC's 0.18 μ CMOS standard cell library. We have implemented both the baseline architecture and a number of enhanced processors consisting of the code integrity checker. With a 1-entry table (register), the area overhead is 2.7%. The overhead for 8-entry and 16-entry tables is 16.5% and 28.8%, respectively. The area overhead is almost linearly dependent on the size of the table. We found that the cycle time of the processor is not changed at all. That is because normally the critical path of a single-issue processor is in the execution stage and our monitoring mechanisms are added in IF and ID stages.

We have also evaluated the performance overhead with a suite of benchmark programs. We assume that the OS handles monitoring exceptions with a least-recently-used (LRU) replacement policy. On each hash miss, the OS replaces half of the entries with hash records from the FHT. We assume

that each OS exception handling takes 100 cycles. Table 1 gives the cycle number overhead for code monitoring. Column 2 reports the total number of execution cycles for the baseline architecture, a single-issue 6-stage PISA processor. Columns 3 and 4 present the number for the enhanced architectures with an IHT of 8 entries and 16 entries, respectively. Columns 5 and 6 are the cycle number overheads. The average clock cycle overhead over the nine applications is 14.7% for 8 entries and 7.7% for 16 entries.

Table 1. Performance overhead

Benchmarks	Clock cycle (10^6)			Overhead (%)	
	No <i>CIC</i>	<i>CIC</i> ₈	<i>CIC</i> ₁₆	<i>CIC</i> ₈	<i>CIC</i> ₁₆
basicmath	158	174.89	159.35	10.7	0.9
susan	25.58	25.63	25.58	0.2	0
dijkstra	54.79	57.6	54.81	5.1	0
patricia	133	146.64	138.81	10.2	4.4
blowfish	37.07	43.32	42.53	16.9	14.7
rijndael	37.6	45.4	37.6	20.7	0
sha	13.21	15.65	13.25	18.5	0.2
stringsearch	4.43	6.65	6.62	50.1	49.4
bitcount	43.62	43.62	43.62	0	0

4 Conclusions

In this paper, we have presented a microarchitectural support for monitoring the integrity of program code running on embedded processors. We select the monitoring level of basic blocks and formulate a mechanism to check the instruction stream within each basic block at run-time. The monitor is incorporated into the processor pipeline seamlessly by augmenting the IF and ID stages of critical instructions with microoperations. The area overhead is reasonable for an IHT with 8 or 16 entries. The maximum frequency from synthesis report does not change at all. The number of execution cycles is slightly increased due to OS exception handling. Our studies reveal that the proposed architecture is capable of detecting a wide range of program code integrity compromises, no matter whether they are caused by security attacks or transient soft errors.

The future work will include refining the entry replacement policy for the IHT to make the methodology more effective and experimenting with other hash algorithms. Meanwhile, it will also be interesting to design the recovery mechanism at either the architectural or OS level.

References

- [1] ASIP Meister. [<http://www.eda-meister.org/asipmeister>].
- [2] ModelSim Simulator. [<http://www.model.com>].
- [3] R. Ragel and S. Parameswaran. IMPRES: Integrated monitoring for processor reliability and security. In *Proc. Design Automation Conf.*, July 2006.
- [4] S. Ravi, A. Raghunathan, P. Kocher, and S. Hattangady. Security in embedded systems: Design challenges. *ACM Trans. on Embedded Computing Systems: Special Issue on Embedded Systems and Security*, 3(3):461–491, Aug. 2004.
- [5] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proc. Int. Symp. on Code Generation & Optimization*, 2005.
- [6] P. Shivakumar, M. Kistler, S. W. Keckler, D. Burger, and L. Alvisi. Modeling the effects of technology trends on the soft error rate of combinational logic. In *Proc. Int. Conf. Dependable Systems & Networks*, pages 389–399, June 2002.