IMPRES: Integrated Monitoring for Processor REliability and Security

Roshan G. Ragel and Sri Parameswaran
School of Computer Science and Engineering, The University of New South Wales &
National Information and Communications Technology Australia (NICTA)
Sydney NSW 2052 Australia
{roshanr,sridevan}@cse.unsw.edu.au

ABSTRACT

Security and reliability in processor based systems are concerns requiring adroit solutions. Security is often compromised by code injection attacks, jeopardizing even 'trusted software'. Reliability is of concern where unintended code is executed in modern processors with ever smaller feature sizes and low voltage swings causing bit flips. Countermeasures by software-only approaches increase code size by large amounts and therefore significantly reduce performance. Hardware assisted approaches add extensive amounts of hardware monitors and thus incur unacceptably high hardware cost. This paper presents a novel hardware/software technique at the granularity of micro-instructions to reduce overheads considerably. Experiments show that our technique incurs an additional hardware overhead of 0.91% and clock period increase of 0.06%. Average clock cycle and code size overheads are just 11.9% and 10.6% for five industry standard application benchmarks. These overheads are far smaller than have been previously encountered.

Categories and Subject Descriptors: B.8.1 [Performance and Reliability]: Reliability, Testing, and Fault-Tolerance

General Terms: Design, Performance, Reliability, Security

Keywords: Detecting Code Injection Attacks, Basic Block Checksumming, Checksum Encryption, Bit Flips Detection

1. INTRODUCTION

Reliability and security of processors have been the subject of extensive research in connection to computing and communications systems. Outcome of reliability and security research includes theoretical improvements in cryptography and security protocols. In addition to theoretical improvements are essential in the area of security, secure implementations are important since security attacks also take advantage of weaknesses in system implementations [1].

Most of the recent security attacks result in demolishing code integrity of an application program by dynamically changing instructions with the intention of gaining access to the program flow. Attacks violating code integrity are called *code injection attacks* as they insert harmful instructions into the dynamic program stream. A detailed survey on code injection attacks is presented in [20]. Guaranteeing code integrity will also provide reliable implementation which detects bit flips, when the hardware is exposed to error-prone environments.

This paper presents a novel hardware integrated technique to deal with code injection attacks and bit flips caused by transient faults. For the first time, we use a technique called the Integrated Monitoring for Processor REliability and Security (IMPRES) to deal with this and show that by handling this problem at the granularity of micro-instructions (MIs) we will be able to reduce the overheads to a considerable minimum. Additionally, we use fault injection experiments to show that IMPRES has full fault detection coverage for bit flips in instruction memory.

The IMPRES framework detects code injection attacks and bit flips in instruction memory. IMPRES will neither detect other security threats, such as second order code injection attacks nor reliability problems, such as bit flips in data memory. A solution for second order code injection attack is described in [5].

The remainder of this paper is organized as follows. A survey on related work is presented in Section 2. Section 3 presents the proposed basic block integrity architecture and admissible program behavior. Section 4 describes a systematic methodology to design the proposed solution for a given application. Results are presented in Section 5 and conclusions in Section 6.

2. RELATED WORK

A wide range of techniques have been suggested in the past to counter code injection attacks. They could broadly be categorized into software based and hardware assisted techniques. Software based techniques use software tools and methods to overcome these attacks without changing the processor architecture. Hardware assisted techniques use architectural support to detect code injection attacks.

Software based techniques could be further categorized into two: one static and the other dynamic. Static techniques try to detect vulnerabilities at compile time. Wagner et al. propose an automated static code analysis tool to detect code that might invite buffer overflow attacks [17], but this produces a relatively large number of false positives. Another static technique uses a language that has only the safe constructs of another language, for example Cyclone [7]. Dynamic software based techniques avoid or considerably reduce code injection attacks at runtime and they either use formal methods to prove a program behaves as expected or use software constructs to monitor proper program behavior at runtime. Proof-Carring Code [10] is an example for the former and Stack Guard [3] is for the later.

Most of the research on hardware assisted techniques concentrate on implementing tamper-resistance and cryptography. In [4], Dyer et al. introduce an IBM co-processor that provides physical tamper-resistance, and hardware support for cryptography. In [14], Ravi et al. investigated the effect of using an embedded processor for similar support. However, hardware assisted techniques are mainly attack-specific. A number of researchers ([8, 9, 19] propose architectural detection support against buffer overflow attack which is one of the code injection attacks.

The ultimate goal of security attacks is to gain control of the system and destroy system integrity by altering information which is in the form of software and data. Embedded Micro Monitoring (EMM) [13] is an architectural framework that uses MI routines to perform in-line security monitoring. EMM performs checking without modifying the application program, and only changing the MIs for selected machine instructions. EMM provides support for reliability at

DAC 2006, July 24–28, 2006, San Francisco, California, USA.

a cost of doubling the memory [12], and partial support for software integrity attacks [13].

Arora et al. [1] use an additional co-processor and hardware tables to perform software integrity checks. They identify program properties at different levels of granularity and store multiple control flow levels of data and checksums to perform software integrity monitoring. Their method produces code which is not relocatable. Our work uses technique similar to [1] in finding program properties for monitoring. However, we have also added one new instruction which contains the checksum of the following basic block. In comparison to [1], our method only needs to check at the basic blocks level, and needs neither additional tables (which may overflow for certain programs) nor complex program analysis.

Thus our **contributions** are: [i] For the first time we have shown a simple hardware/software method for monitoring code injection attacks which requires only a rudimentary software analysis; [ii] This monitor is also capable of picking up transient faults such as bit flips. We have used a fault injection engine to show the fault coverage of this technique; and [iii] Produces code which is relocatable allowing the use of this technique with an operating system.

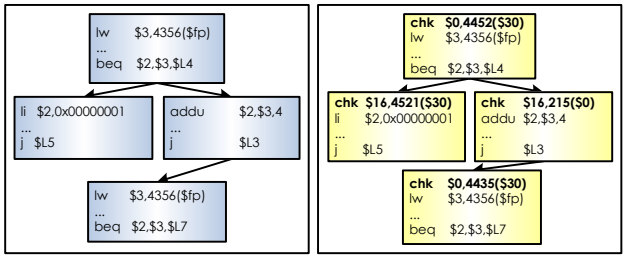
IMPRES for the first time is a generic, scalable, low overhead method to both improve reliability and protect against code injection attacks. Software methods only detect bit flips and are susceptible to code injection attacks [18]. Hardware methods use tables [1] or watchdog processor's memory which are neither scalable nor relocatable. Ours overcomes these shortcomings. Additionally, by the use of encryption, we protect against an attacker changing a whole basic block with the checksum, to gain access to the executable. We believe that our contributions will make practical the deployment of software integrity checkers for real applications.

3. ARCHITECTURE

In this section, we give an overview of the monitoring architecture and discuss the admissible application behavior when this monitor is present in a processor. We argue that the runtime code integrity could be fully preserved if we ensure that all the basic blocks of an application program are intact. Thus performing only basic block integrity checks is sufficient to ensure software integrity.

3.1 Basic Block Integrity Architecture

The proposed basic block integrity checker incorporates the following three tasks: (1) identifying basic blocks and calculating and assigning checksums for each basic block at compile time; (2) encrypting the checksums with a secret hardware key at load time; and (3) re-calculating the encrypted checksums at runtime and comparing the encrypted checksums with loaded values.



(a) CFG for the code segment (b) CFG with check instructions

Figure 1: Compile time instrumentation of the application

Figure 1 illustrates the compile time instrumentation for a given code segment. A code segment grouped into basic blocks based on the control flow is depicted in Figure 1(a). Then, each basic block is processed separately to calculate a checksum based on the instructions of that block. The calculated checksum is then inserted at the beginning of each basic block using a special instruction (*chk* instruction) as in Figure 1(b).

Figure 2 depicts how a basic block with a checksum is securely loaded into memory, and how a hardware integrated monitor is used to detect code integrity violations at runtime. A basic block with calculated checksum is loaded using a secure loader. While loading instructions, the loader will use a hardware key (which is randomly generated for each loading) to encrypt the calculated checksum into an encrypted checksum.

Right half of Figure 2 depicts how a code integrity violation is captured at runtime. The first instruction of a loaded basic block is a *chk* instruction that carries the encrypted checksum for the corresponding basic block. When an instruction of this kind is fetched, the encrypted checksum is loaded into a special register (*eChkSum*). Checksum for each basic block is incrementally re-calculated at runtime while instructions belonging to the basic block are executed and is stored in another special register (*iChkSum*). The incremental re-calculation is achieved using MIs integrated into each machine instruction. Last instruction of each basic block is a control flow instruction (CFI) and if not, one is inserted (if it is not present at the end of a basic block). MIs for CFIs are altered such that they will (a) encrypt the incrementally stored checksum with the same hardware key used during loading and (b) compare the result against the one loaded from *chk* instruction. A mismatch in the comparison will indicate a code integrity violation and generate a *SIGCKSM* signal.

Apart from encrypted checksums, a special single bit flag (*fBB*) is used to capture code integrity violations those escape the encrypted checksum technique. When a program is loaded or a CFI is executed, *fBB* is set. When a non CFI is executed *fBB* is cleared. When a *chk* instruction is executed and if *fBB* is not set (this occurs when the CFI of the last basic block is not executed) then a 'no CFI' error is signaled (*SIGNCFI*).

Our technique hugely differs from other checksum- or hash-based software integrity checking techniques, where hashing is performed at the beginning or end of basic blocks and therefore accumulates the workload to particular points in the program flow. IMPRES distributes the overhead to all the instructions, thus the total hardware related overhead is reduced to an amount that is negligible. Encryption algorithms are complex and incur high latency, while calculating checksums are less complex. Therefore we have used checksumming together with encryption to perform just a few encryptions while ensuring that the technique is still secure.

3.2 Admissible Application Behavior

A number of determinants could be considered when choosing program properties to be monitored. Importantly the properties chosen should clearly indicate when a violation of behavior occurs. As we have already discussed, we will only be ensuring integrity of basic blocks.

After compile time instrumentation, each basic block (BB) will start with a *chk* instruction and end with a CFI. We will call these two instructions boundary instructions (BI) and all other instructions in a basic block - non boundary instructions (non-BI). If we can assure that each and every basic block in an application is intact, then we can safely assume that the code integrity is enforced for the whole application. We classify the possible code integrity violations due to code corruption or code injection attack under different categories as shown in Table 1. The first column in Table 1 names different types of code integrity violations. Column two, represents the original instructions and three the corrupted/injected instructions for a BB. The forth column in Table 1 shows the error signals related to each code integrity violations.

Type	Original	Changed	Error Signals
T1	a non-BI	another non-BI	SIGCKSM
T2	a non-BI	a chk instruction	SIGNCFI
T3	a non-BI	a CFI	SIGCKSM
T4	chk instruction	change checksum	SIGCKSM
T5	chk instruction	a CFI	SIGCKSM
T6	chk instruction	a non-BI	SIGCKSM
T7	CFI	another CFI	SIGCKSM
T8	CFI	changed target	SIGCKSM
T9	CFI	a non-BI	SIGNCFI
T10	the whole BB	another BB	SIGCKSM/SIGNCFI

Table 1: Different Types of Code Integrity Violations

As shown in Table 1, types T1, T3, T4, T5, T6, T7 and T8 violations generate *SIGCKSM* signals. In T1, the runtime checksum is going to be in error and therefore will not match the loaded value. This will be detected when the CFI at the end of the BB is executed and a *SIGCKSM* signal is raised. In a violation of type T3, the checksum comparison is going to be performed by the corrupted instruction and

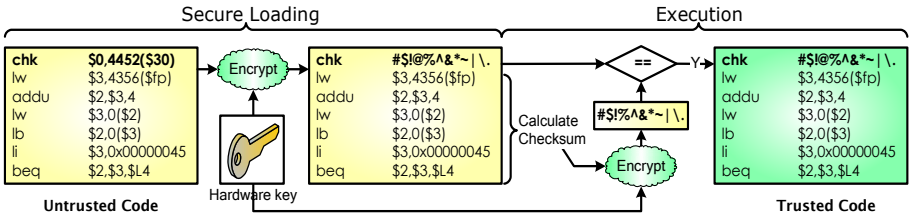


Figure 2: Secure Loading and Execution

this will result in a comparison mismatch and therefore a *SIGCKSM* signal. When the loaded checksum of the *chk* instruction is changed (as in T4), the checksum comparison at the end of the BB will generate a mismatch and *SIGCKSM* signal. When a *chk* instruction is converted into a CFI (as in T5) a checksum mismatch will occur and this will result in a *SIGCKSM* signal. When a *chk* instruction is changed into a non-BI instruction (as in type T6), the following CFI will raise a *SIGCKSM* signal. A violation of type T7 will result in a faulty runtime checksum as the checksumming includes CFI. This will again result in a checksum mismatch and therefore will generate *SIGCKSM* signal. When the target of the CFI is changed (as in T8), the runtime checksum is going to be in fault and therefore will generate a *SIGCKSM* signal.

Code integrity violations of type T2 and T9 are going to generate *SIGNCFI* signals. When a non-BI is changed into a *chk* instruction (as in T2) there are going to be two consecutive *chk* instructions in the program flow and this is going to generate a *SIGNCFI* signal. A violation of type T9 will result in an execution of a *chk* instruction (from the next BB) without the last CFI being executed in the current BB. Therefore, a *SIGNCFI* signal will be generated.

Type T10 of code integrity violation occurs when a whole BB being replaced by a fake BB. A fake BB might contain: *[case1]* one or more CFIs; *[case2]* one or more *chk* instructions; and *[case3]* only non-BIs. CFIs in a fake BB (as in case1) will cause checksum mismatch at the first CFI and generate *SIGCKSM* signal. When the fake BB has one or more *chk* instructions (as in case2), if the program flow reaches the fake BB via a non-CFI then a *SIGNCFI* signal will be raised at the first *chk* instruction, otherwise based on the next BI either a *SIGCKSM* or a *SIGNCFI* signal will be raised. When there are no BIs in the fake BB, depends on the next BI, either a *SIGCKSM* signal or a *SIGNCFI* signal will be generated. A fake BB to act as a genuine BB it should contain proper BIs. But generating a valid *chk* instruction is prevented by the encryption technique described in Figure 2. Since it is impossible for an intruder to get hold of the randomly generated secret hardware key, the attacker will not be able to generate a valid BB from injected or malicious code. In the worst case, if an attacker has managed to crack the hardware key for a particular process, it is impossible to perform a mass attack as each processor will have random hardware keys, which cannot be guessed.

4. DESIGN FLOW

In this section, an overview of the proposed design flow for the checking architecture is provided. First, the design of a software interface that allows the applications to interact with the architectural enhancement is described, and then the design of the architectural enhancement itself is discussed.

4.1 Software Design

Left half of Figure 3 describes the implementation details of the interface between an application program and monitoring hardware. It is worth noting that check instructions inserted at the beginning of BBs and MIs embedded into machine instructions serve as the interface between software and hardware. In software instrumentation process, the source code of an application is compiled by the front end of a compiler and the assembly code for the target Instruction Set Architecture (ISA) is produced. Then a software parser is used to instrument the assembly code. Tasks performed by this parser are: [i] identifying basic blocks; [ii] calculating checksums for each basic block based on the instruction sequence of each basic block; [iii] forming check instructions by inserting the checksums calculated in tasks 2 above; and [iv] inserting the *chk* instructions into the assembly instruction stream at the beginning of each basic block.

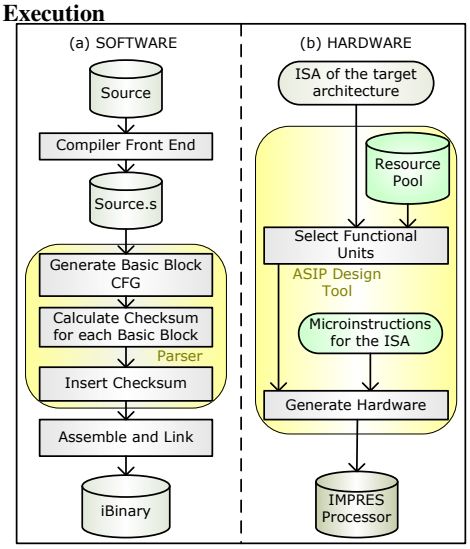


Figure 3: Design flow for the monitoring architecture

An instrumented version of the assembly program is then assembled and linked using the back-end of the same compiler to generate the instrumented binary (indicated by iBinary in Figure 3) for the target architecture. The loader of the same compiler tool kit is modified to integrate secure loading as described in Figure 2.

4.2 Architectural Design

Without losing generality of our technique, we use an automatic processor design tool to implement it in hardware. This automatic design tool is used to design Application Specific Instruction-set Processors (ASIPs). Right half of Figure 3 describes this design process. The ASIP design tool allows us to write MIs for each machine instruction and to add new machine instructions. We used this feature to add the new machine instruction called *chk*. We amend all CFIs to perform encryption and comparison as illustrated in Figure 2. We also modify MIs for non-BI instructions so that they will perform instruction checksumming. The reader may refer to [13] for the details on writing MIs in the design tool that we used.

The final task in the architectural design process is to generate the hardware model in a hardware description language for simulation [behavioral model] and synthesis [gate level model]. The same hardware monitoring is re-implemented in a cycle accurate instruction set simulator for performance evaluation and fault injection analysis.

5. EXPERIMENTAL RESULTS

In this section, we present the hardware overhead incurred by the proposed architecture, as well as the impact of the technique on performance. For the purpose of experiments, we have used PISA instruction set (as implemented in SimpleScalar tool set [2]) for building a secure processor from the one in [11]. Applications from MiBench [6] benchmark suite are used in the experiments. We have also performed fault injection analysis for single instruction corruption.

5.1 Evaluating the Overheads

To evaluate the methodology, applications are compiled with the GNU/GCC® cross compiler for PISA instruction set as described in Section 4.1. An automatic ASIP design tool, ASIP Meister [15] is

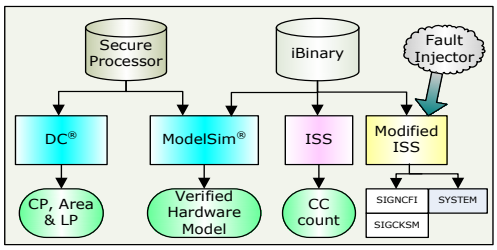


Figure 4: Testing and Evaluation

used to generate the VHDL description of the target processor as described in Section 4.2. The outputs of ASIP Meister are the VHDL models for simulation and synthesis of the secure processor. The synthesis model is used with Synopsys Design Compiler® (DC) [16] (as shown in Figure 4) to obtain the area, clock period (CP) and the leakage power (LP). TSMC's 90nm core library is used for synthesis with typical conditions enabled. ModelSim® hardware simulator is used with the instrumented binary (iBinary) to verify the correctness of the monitoring hardware.

	Clock Period (ns)	**Area** (# of cells)	**Leakage Power** (μW)
Ordinary H/W	16.84	227077	478
IMPRES H/W	16.85	229143	483
Overhead(%)	0.06%	0.91%	1.05%

Table 2: Clock Period, Area and Leakage Power Comparison

Table 2 shows the clock period, area and leakage power overheads of our processor due to extra monitoring logic implementation. The clock period has increased a negligible 0.06% and the area increase is a paltry 0.91%. Leakage power estimation shows an increase of 1.05%.

Applications	Clock Cycle (10^6)			Code Size (# of lines)		
	No chk	With chk	Inc. (%)	No chk	With chk	Inc. (%)
adpcm.encode	74.4	89.8	17.2	402	460	14.4
adpcm.decode	57.4	69.8	17.7	397	452	13.9
blowfish.encrypt	58.3	62.6	6.74	2946	3085	4.72
blowfish.decrypt	57.0	63.2	6.66	2946	3085	4.72
crc32.checksum	42.5	48.0	11.4	527	607	15.2

Table 3: Performance and Code Size Overheads

As shown in Figure 4, the binary produced from software design is used with an instruction set simulator (ISS) to compute the clock cycle (CC) overhead of our design due to the software instrumentation. Table 3 shows CC and code size overheads due to additional *chk* instructions in the runtime instruction stream. The bigger the basic blocks in an application, the smaller the percentage of CC and code size overheads, as each basic block carries an additional *chk* instruction. The *blowfish* application has bigger basic blocks compared to others. This explains the lower percentage of CC and code size overheads of *blowfish* application. Average clock cycle and code size overheads over the five applications are 11.9% and 10.6% respectively.

5.2 Fault Injection Analysis

We performed fault injection analysis by altering the SimpleScalar tool set and adding fault injection scripts to the tool set as shown by the right side of Figure 4. Note that this fault injection analysis only checks for reliability and not security.

The fault injection analysis is performed as follows: (1) after an application is loaded, a random address within the memory address boundaries is generated; (2) the instruction at this memory address is changed/corrupted with a randomly generated instruction or bit flipping; and (3) the application is executed and the output trace is checked for fault activation and detection.

Table 4 shows the results of the fault injection analysis. It is performed 10000 times for each application and results are tabulated. The first column of Table 4 names the applications used for this analysis. Not all the faults injected are activated as some of them may fall into the non execution path of the program. These faults are called not activated (*Not Act.*). Columns 3,4 and 5 of Table 4 indicate the detected faults by *System* (faults detected by the ISS itself - for example, an invalid opcode is detected by the ISS) *SIGCKSM* and *SIGNCFI* respectively. Most of the activated faults are detected by our checksumming technique (*CHSM*) and the system detects the second largest amount. From Table 4 it is evident that all single instruction corruptions or changes are captured either by our monitor or by the system.

Applications	Not Act.	System	CKSM	NCFI	Total
adpcm.encode	2032	434	7396	138	10000
adpcm.decode	1595	414	7857	134	10000
blowfish.encrypt	4460	242	5233	65	10000
blowfish.decrypt	4314	267	5365	54	10000
crc32.checksum	4960	276	4650	114	10000

Table 4: Fault Injection Results

6. CONCLUSIONS

In this paper, we have for the first time presented a simple hardware-assisted runtime technique to detect code integrity violations. We have formulated a list of acceptable application behavior and evaluated the fault coverage of our system via fault injection analysis. We have elaborated an automatic technique to design our solution using an ASIP design tool and software instrumentation. The hardware overhead and clock period change are evaluated and reported. The results show that these overheads are very small and negligible. Our study reveals that the proposed IMPRES framework is capable of handling code injection attacks and transient bits flip via detecting code integrity violations with minimal overheads. We conclude that our technique is useful in answering the increasing security and reliability concerns in computer systems.

7. REFERENCES

[1] D. Arora et al. Secure embedded processing through hardware-assisted runtime monitoring. In *Proceedings of the DATE'05*, volume 1, 2005.

[2] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3):13–25, 1997.

[3] C. Cowan et al. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proc. 7th USENIX Security Conference*, pages 63–78, San Antonio, Texas, 1998.

[4] J. G. Dyer et al. Building the ibm 4758 secure coprocessor. *Computer*, 34(10):57–66, 2001.

[5] Gunter Ollmann. Second-order Code Injection Attacks: Advanced Code Injection Techniques and Testing Procedures, 2003.

[6] M. R. Guthaus et al. Mibench: A free, commercially representative Embedded Benchmark Suite. *In IEEE 4th Annual Workshop on Workload Characterization, Austin, TX*, pages 83–94, December 2001.

[7] T. Jim et al. Cyclone: A Safe Dialect of C. In *In USENIX annual technical conference, Monterey, CA, June 2002.*, 2002.

[8] R. Lee et al. Enlisting hardware architecture to thwart malicious code injection. In *Proceedings of the International Conference on Security in Pervasive Computing*. Springer Verlag LNCS, March 2003.

[9] J. McGregor et al. A processor architecture defense against buffer overflow attacks. In *Proceedings of the SPC'03*, pages 237–252. Springer Verlag, March 2003.

[10] G. C. Necula. Proof-Carrying Code. In *Proceedings of POPL'97*, pages 106–119, Paris, France, Jan 1997.

[11] J. Peddersen et al. Rapid Embedded Hardware/Software System Generation. In *18th International Conference on VLSI Design held jointly with 4th International Conference on Embedded Systems Design (VLSID'05)*, pages 111–116, January 2005.

[12] R. G. Ragel and S. Parameswaran. Soft error detection and recovery in application specific instruction-set processors. In *Proceedings of the SELSE-1*, April 2005.

[13] R. G. Ragel, S. Parameswaran, and S. M. Kia. Micro Embedded Monitoring for Security in Application Specific Instruction-set Processors. In *Proceedings of the CASES'05*. ACM Press, September 2005.

[14] S. Ravi et al. Security in embedded systems: Design challenges. *Trans. on Embedded Computing Sys.*, 3(3):461–491, 2004.

[15] The PEAS Team. ASIP Meister, http://www.eda-meister.org/asip-meister/, 2002.

[16] The Synopsys Team. Synopsys Design Compiler, The industry standard for logic synthesis. http://www.synopsys.com/.

[17] D. Wagner et al. A first step towards automated detection of buffer overrun vulnerabilities. In *Network and Distributed System Security Symposium*, pages 3–17, San Diego, CA, February 2000.

[18] G. Wurster, P. C. van Oorschot, and A. Somayaji. A generic attack on checksumming-based software tamper resistance. In *SP '05: Proceedings of the 2005 IEEE Symposium on Security and Privacy*, pages 127–138, Washington, DC, USA, 2005. IEEE Computer Society.

[19] J. Xu et al. Architecture support for defending against buffer overflow attacks. In *EASY-2 Workshop*, October 2002.

[20] Y. Younan, W. Joosen, and F. Piessens. Code injection in C and CPP: A Survey of Vulnerabilities and Countermeasure, July 2004.