Hardware Support for Code Integrity in Embedded Processors

Milena Milenković
WBI Performance II
IBM
Austin, TX 78758
milena@computer.org

Aleksandar Milenković
Emil Jovanov
Electrical and Computer Engineering Department
The University of Alabama in Huntsville
Huntsville, AL 35899
{milenka, jovanov}@ece.uah.edu

ABSTRACT

Computer security becomes increasingly important with continual growth of the number of interconnected computing platforms. Moreover, as capabilities of embedded processors increase, the applications running on these systems also grow in size and complexity, and so does the number of security vulnerabilities. Attacks that impair code integrity by injecting and executing malicious code are one of the major security issues. This problem can be addressed at different levels, from more secure software and operating systems, down to solutions that require hardware support. Most of the existing techniques tackle the problem of security flaws at the software level, but this approach lacks generality and often induces prohibitive overhead in performance and cost, or generates a significant number of false alarms. On the other hand, a further increase in the number of transistors on a single chip enables integrated hardware support for functions that formerly were restricted to the software domain. Hardware-supported defense techniques have the potential to be more general and more efficient than solely software solutions. This paper proposes four new architectural extensions to ensure complete run-time code integrity using instruction block signature verification. The experimental analysis shows that the proposed techniques have low performance and energy overhead. In addition, the proposed mechanism has low hardware complexity, and does not impose either changes to the compiler or changes to the existing instruction set architecture.

Categories and Subject Descriptors

C.1 [**Processor Architectures**]: Miscellaneous;
C3. [**Special-purpose and application-based Systems**]: Real-time and embedded systems;
K6. [**Management of Computing and Information Systems**]: Security and Protection.

General Terms

Performance, Design, Security.

CASES'05, September 24–27, 2005, San Francisco, California, USA.

Keywords

Code integrity, code injection attacks.

1. INTRODUCTION

With the exponential growth of the number of interconnected computing platforms, computer security has become a critical issue. The utmost importance of system security is further underscored by the expected proliferation of diverse Internet-enabled embedded systems — ranging from home appliances, cars, and sensor networks to personal health monitoring devices.

One of the security issues that have recently drawn attention is software integrity, which ensures that the executing instructions have not been changed by either an accident or an attack. The attacks impairing software integrity are called *code injection attacks*, since they inject and execute malicious code instead of regularly installed programs. The most widely known type of code injection attacks is so-called *stack smashing* [31]. Figure 1 illustrates one such attack for an architecture where the stack grows towards lower memory addresses. A function accepts untrustworthy values into a local buffer *Buf*. If the function does not verify whether the length of the input exceeds the buffer size, an attacker can easily overflow the buffer. By overflowing the buffer, any location on the stack in the address space after the beginning of the buffer can be overwritten, including the return address of the vulnerable function. Using this mechanism, an attacker can insert malicious code sequence, and overwrite the return address to point to the malicious code. Other attacks may overflow buffers stored on the heap [8] or exploit integer errors [3], dangling pointers [13], or format string vulnerabilities [30]. Most programs with these vulnerabilities are also susceptible to so-called *return-into-libc* attacks, where an attacker changes a code pointer to point to the existing code, usually the library code. Return-into-libc attacks are also called arc injection, since they inject an arc in a control flow graph of a program.

The number of reported software vulnerabilities has grown from 171 in 1995 to 4,129 in 2002, according to the United States Computer Emergency Readiness Team Coordination Center (US-CERT/CC) [39]; a large number of these vulnerabilities can be exploited by code injection attacks. Although most reported vulnerabilities are found in programs executing on desktop and server platforms, the increasing complexity of embedded system applications may result in similar software flaws. For example, one recent Cyber Security Bulletin from US-CERT reports multiple buffer overflow vulnerabilities in a Bluetooth connectivity program for Personal Digital Assistants (PDAs) [40].

As the communication and computation capabilities of smart phones, PDAs, and other embedded systems continue to grow, so will the number of malicious attacks.

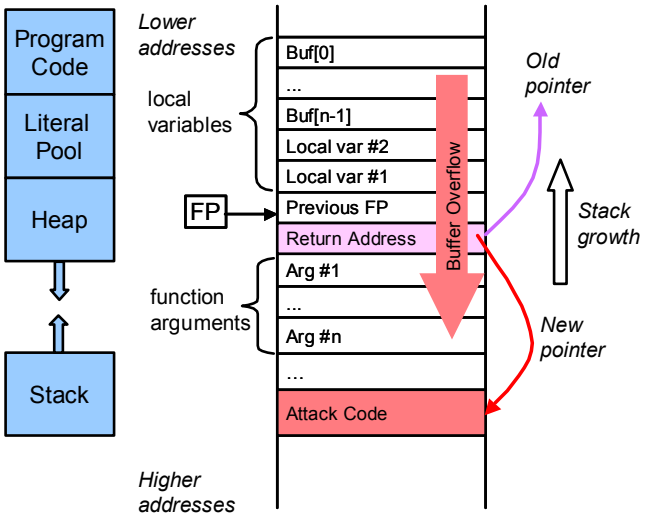

Figure 1. An illustration of the stack smashing attack.

The multitude of code injection attacks prompted development of a large number of predominantly software-based counter-measures. Static software techniques rely on formal analysis and/or programmers' annotations to detect security flaws in the code, and then leave it to the programmers to correct these flaws [14, 41]. However, using these techniques has yet to become a common programming practice. Moreover, they fail to discover all vulnerabilities, suffer from false alarms, and put an additional burden on programmers. On the other hand, dynamic software techniques augment the original code or operating system to detect malicious attacks and to terminate attacked programs, or to reduce the attacker's chances of success [5, 7, 10, 11, 16, 21, 24, 32, 34]. Though effective, these techniques can result in significant performance overhead and usually require program recompilation, so they are not readily applicable to legacy software.

Several recent research efforts propose hardware-supported techniques to prevent unauthorized changes of program control flow [9, 12, 15, 18, 20, 23, 28, 29, 33, 35, 37, 38, 43, 45]. These techniques have potential to offer better protection from code injection attacks with lower performance and energy overhead than techniques relying solely on software. However, most of hardware-supported techniques focus only on stack smashing, or still have a significant performance overhead, or do not thoroughly explore the implications of implementation choices. We believe that there is a need for a new hardware security layer to prevent the whole class of code injection attacks.

In this paper we propose and evaluate new architectural extensions to ensure run-time code integrity at minimal cost, energy overhead, and performance loss. The proposed techniques share a common mechanism: Instruction blocks are signed using secret hardware keys during a secure program installation process, and signatures are stored with the code. During secure program execution, signatures are recalculated from instructions and compared to the stored signatures. If the two values do not match, the program cannot be trusted and should be terminated. The proposed mechanism does not require significant processor changes; it is cost-effective and requires no changes in legacy source code and no compiler support. In addition, encrypted instruction block signatures protect the code from software tampering, and enable fault detection in error-prone environments such as space. Though the proposed mechanism offers protection from code injection attacks only, it can be easily expanded to protect from return-into-libc attacks.

The results of detailed cycle-by-cycle simulations indicate that the signature verification mechanism with hardware support indeed does have low performance and energy overhead. In an embedded processor with a 4K-instruction cache (I-cache), the average performance overhead of the best low-complexity technique is 2.6%, and the average energy overhead is 8%. With additional hardware investment in a signature cache, the performance and energy overhead can be reduced to 0.5% and 3%, respectively.

The rest of this paper is organized as follows. Section 2 describes the proposed architectures for instruction block verification. Section 3 describes the experimental methodology, and Section 4 discusses the results of the performance and energy analysis. Section 5 describes the related work and the last section concludes the paper.

2. ARCHITECTURES FOR INSTRUCTION BLOCK VERIFICATION

The proposed techniques share a common sign-and-verify mechanism, where instruction blocks are signed during the secure installation process, and signatures are verified during secure program execution for each instruction block fetched from memory (Figure 2). Four techniques discussed differ in signature placement (embedded in the code vs. separate code segment) and in signature handling after verification (discard vs. keep in a signature cache).

2.1 Secure Installation

The role of the secure installation process is to generate signatures for programs whose integrity we want to protect, and to store them together with the program binary. The algorithm for signature generation should satisfy two opposing requirements: On the one hand, signatures should be cryptographically secure; i.e., it should be very hard for an attacker to generate a correct signature for a given instruction block. On the other hand, signature verification should not add a significant overhead to program execution. An additional requirement is that the signature mechanism should require minimal or no compiler support, so that any already-compiled code can be securely installed. To satisfy these requirements, we decided to use a combination of an extended version of a Multiple Input Shift Register (MISR) and Advance Encryption Standard (AES), although other algorithms may also be used.

A signature of an instruction block is obtained in the following way (Figure 2). First, all instructions in that block pass through a MISR. The MISR is essentially a shift register with linear feedback coefficients, so a new value of the MISR is a function of the current value and incoming instructions. After each instruction block, the MISR is initialized to a predefined start value. Linear feedback connections and the start value are determined by secret

processor keys hidden in permanent read-only registers. The only program that can read these registers is the secure installer, which is protected by password or smart card. This installation process is similar to the one proposed by Kirovski et al. [20].

An attacker could discover the MISR secret keys if he/she manages to read the stored signatures, and compare them to the corresponding instruction blocks. To prevent successful read attacks, the result of the final MISR calculation is encrypted using AES, which is proved to be secure. The AES key is also stored in a read-only protected register and thus is hidden from attackers. With the signature length of 128 bits, a brute force buffer overflow attack would need to overflow the buffer up to 2^{128} times to find an instruction block that is accepted by the system. With the possibility of a buffer overflow each second of program execution, an attacker would need more than 10^{31} years for a successful attack.

To achieve signature generation without any compiler support, an instruction block is defined as a block of k consecutive instructions, where k is fixed for a given computer system. Hence, the secure installation simply breaks up executable sections of a program binary to instruction blocks, and signs each block separately. If the last block is shorter than k, it is padded with randomly generated instructions that do not change the state of the processor.

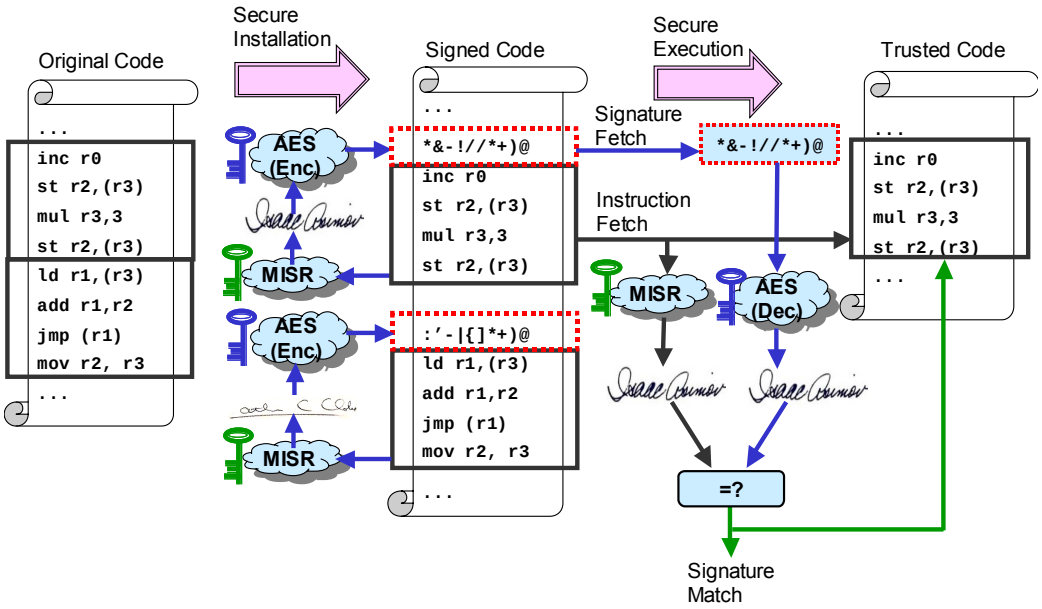

Figure 2. Mechanism for trusted instruction execution.

2.2 Secure Program Execution

When an instruction is fetched from memory, the integrity of the corresponding instruction block needs to be verified, so the whole instruction block must be fetched. Consequently, the most suitable instruction block size is the size of the lowest level of the instruction cache (the cache that is the closest to memory), or the size of the fetch buffer in systems without the cache. Without loss of generality, in the rest of this paper, we focus on a system with separated data and instruction first level caches, and no second level cache. The instruction cache (I-cache) is a read-only resource, so the integrity is guaranteed for instructions already in the I-cache. Hence, signatures need to be verified only on I-cache misses.

Signatures are verified using a dedicated hardware resource called the Instruction Block Signature Verification Unit (IBSVU, Figure 3). The IBSVU encompasses registers for buffering instructions and signatures, support for AES decryption, MISR, and control logic. On an I-cache miss, the corresponding signature is fetched before instructions and temporarily stored in an IBSVU register called SIGM. Note that the signatures are not stored in the cache memory, since they are not needed for blocks already in the I-cache.

Fetched instructions pass through the MISR register with the linear coefficients that are equal to those used during secure installation. Concurrently with MISR calculation, the AES block decrypts the signature from the SIGM register. Hence, the decryption time is partially or completely overlapped with the instruction block fetch phase. The decrypted signature is compared to the final MISR calculation: If the two values match, the instruction block is properly installed and can be trusted. If the values differ, the instruction block includes injected code or it is not properly installed, so a trap to the operating system is asserted. The operating system then aborts the process whose code integrity cannot be guaranteed and possibly audits the event.

An embedded computing system might be designed to run only in the secure execution mode. However, if an application does not accept input data from untrustworthy channels, a user might decide to install it without instruction block signatures. Such an application then executes in the ordinary, insecure mode. The information about the required execution mode could be added to the program header during installation.

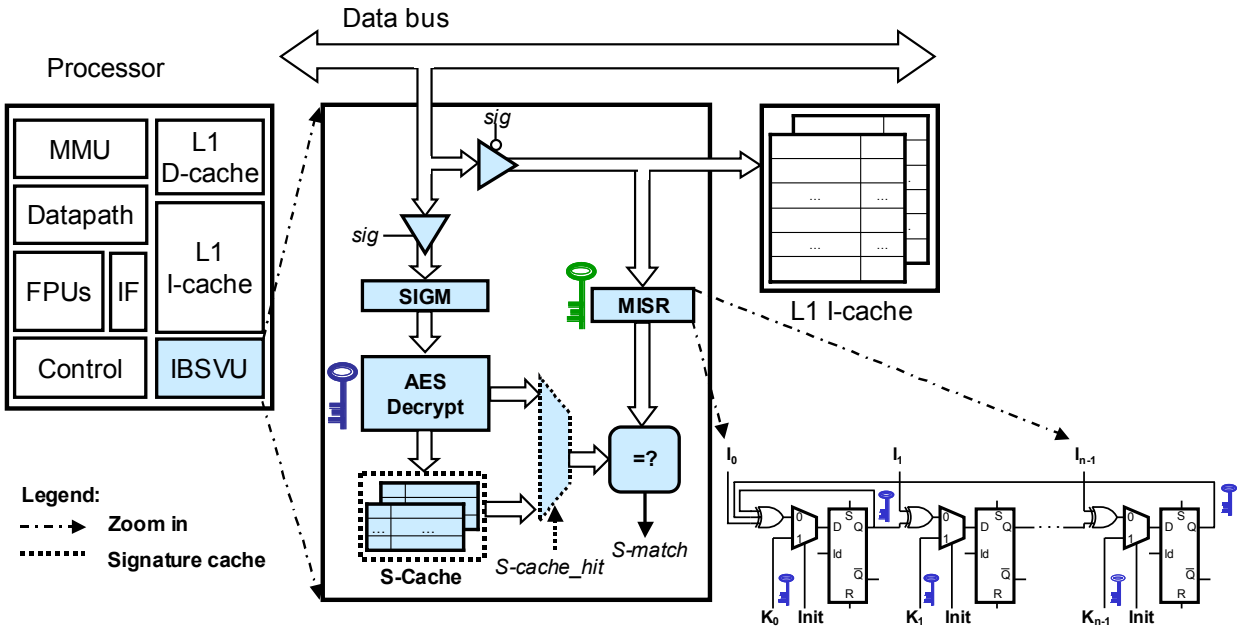

Figure 3. Hardware support for signature verification.

2.3 Implementation Details

We propose four techniques based on the common sign-and-verify mechanism: SIGCED, SIGCEK, SIGCTD, and SIGCTK. These techniques can be classified according to the signature placement in the code and signature handling after verification. Instruction block signatures can be stored before the corresponding instruction blocks, i.e., embedded in the code (SIGC*E*D, SIGC*E*K). Another possibility is to store all signatures in a separate code section that we call the signature table (SIGC*T*D, SIGC*T*K). After verification, a signature can be discarded (SIGCE*D*, SIGCT*D*), or kept in a decrypted form in a dedicated read-only resource called the signature cache – S-cache (SIGCE*K*, SIGCT*K*), shown with dotted lines in Figure 3. The S-cache can be accessed only by the IBSVU. The S-cache's number of entries and organization differ from those of the I-cache in order to keep decrypted signatures even when the corresponding instruction blocks are evicted from the I-cache. The S-cache may reduce the overhead of run-time signature verification at the price of increased hardware complexity. The potential benefit of an S-cache is twofold: first, if a signature is found in the S-cache, it does not have to be fetched from memory, thus saving both processor cycles and energy. Second, since signatures in the S-cache are decrypted, an S-cache hit also avoid power overhead for AES decryption. However, a relatively large S-cache may significantly increase processor area, thus increasing total power dissipation.

Figure 4 illustrates the modifications in the instruction fetch control flow for the SIGCED. The value of the program counter (PC) is used to access the I-cache. Note that without loss of generality, we assume that the I-cache is indexed by virtual addresses and it is virtually tagged. This is a frequent case in embedded processor caches, for example in Intel's Xscale processor [2]. In the case of a cache hit, the instruction is fetched from the I-cache and there is no need for signature verification. In the case of a cache miss, we need to calculate the address of the instruction block to be fetched in the virtual memory. The instruction block address has changed because of signature embedding and added padding. The secure installation process adds padding to the code with embedded signatures so that no instruction block is split between two pages. When a correct virtual address is calculated, the translation look-aside buffer (TLB) is accessed for virtual to physical address translation. The signature is fetched before instructions and decrypted while instructions are being fetched. With the SIGCEK technique, an I-cache miss is followed by the corresponding S-cache lookup. Hence, the signature needs to be fetched and decrypted only if it is not found in the S-cache.

Figure 5 illustrates modifications in the instruction fetch control flow with SIGCTD. The address of a signature *SigAddress* is calculated in parallel with the I-cache lookup, and it must not be greater than the *SigTableEnd*, the address of the last signature in the signature table. With the SIGCTK technique, the signature fetch and decrypt phases may be avoided if a signature is found in the S-cache.

The SIGCTD and SIGCTK techniques have both advantages and disadvantages over techniques with embedded signatures. Since signatures are stored separately from instructions, there is no need for padding and hardware address translation. On the other hand, signature fetch from memory requires a completely separate memory access. If the application code is relatively large, instructions and signatures may even be located on separate pages, so accesses to signatures may cause page faults.

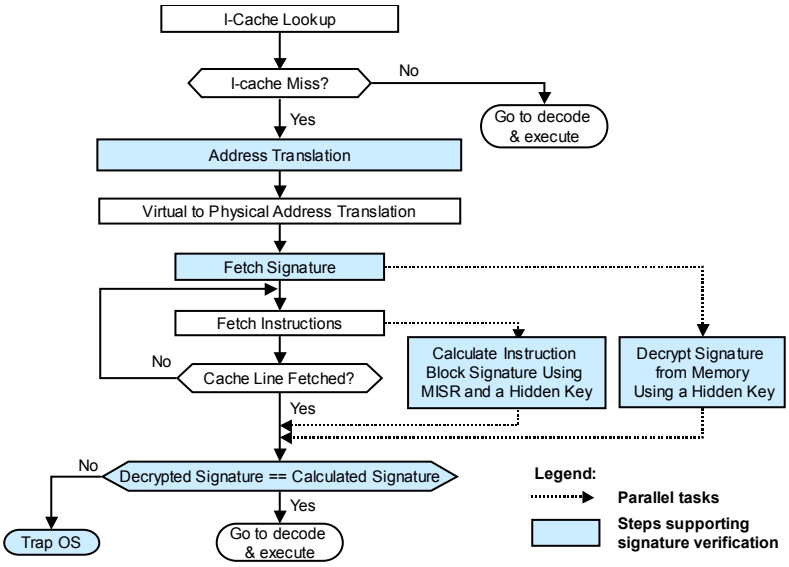

Figure 4. SIGCED: Signature verification control flow.

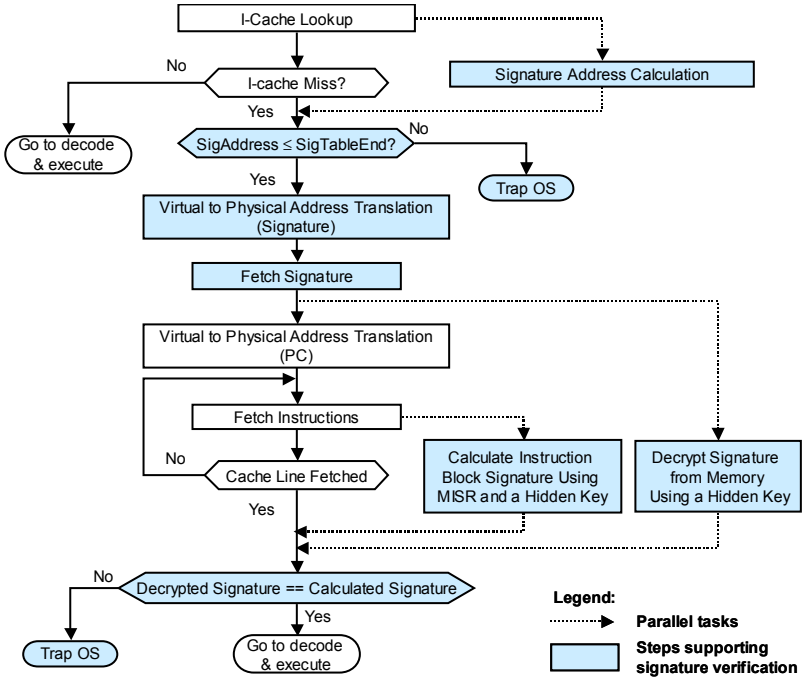

Figure 5. SIGCTD: Signature verification control flow.

2.4 Other Considerations

With the proposed techniques, each dynamically linked library (DLL) has its own signature section or embedded signatures, so all code can be safely verified. The pointer to the signature section or the beginning of the code with embedded signatures can be loaded to the IBSVU when a particular library is dynamically linked. The IBSVU stores a fixed number of such pointers. When an application is dynamically linked with more DLLs than the IBSVU can hold, the overflow is handled by the operating system, and the overflow data is stored in memory.

Another consideration is dynamically generated code, such as the code generated by the Java Just-In-Time compiler, which may never be saved in an executable file. Such code can be marked as non-signed and executed in the unprotected mode, or the code generator can generate the signatures together with the code. If the generator is trusted, its output should be trusted too. The same argument applies to the interpreted code.

Although effective against all injection attacks of malicious code, the described signature verification mechanism cannot detect a *return-into-libc* attack. Signatures embedded in the code also do

not prevent an attack where the injected code consists of copies of regularly installed instruction blocks and corresponding signatures (a form of replay attack). However, the basic mechanism can be expanded to defend against such attacks. Replay attacks can be easily prevented if an instruction block signature is a function of not only instructions, but also the relative offset of that block from the beginning of the code: e.g., if the relative address of the first instruction in a block also passes through the MISR during secure installation. Unauthorized jumps into existing code can be prevented by embedding allowed target addresses in signatures, similar to branch address hashing [36]; unauthorized returns can be disabled by combining the signature verification with a form of the secure stack.

3. EXPERIMENTAL METHODOLOGY

A hardware-supported defense technique should not add significant overhead in hardware complexity, execution time, energy consumption, and memory requirements. A qualitative assessment indicates relatively low hardware complexity of techniques without the S-cache; the complexity of techniques with the S-cache depends on the S-cache size. We evaluate only implementations where the S-cache is smaller than the I-cache, so the complexity of these techniques is also moderate.

The memory overhead is simply determined by comparing the sizes of the original code and the code with signatures. To emulate the secure installation process, we have developed a program that calculates signatures of instruction blocks in executable sections of programs in the ELF format, and modifies programs to include calculated signatures [27]. The signature size is 128 bits, a minimum size for AES encryption. We have chosen the MISR coefficients to be the coefficients of a primitive polynomial of the 128-th order.

The performance overhead is evaluated using a modified SimpleScalar ARM simulator [4] that supports the SIGCED, SIGCEK, SIGCTD, and SIGCTK techniques. As a measure of performance we use the average number of cycles per instruction (CPI) and compare the CPI of the proposed techniques to the CPI of the Base configuration (without signature verification).

In order to evaluate the proposed techniques' sensitivity to different system configurations, we varied the I-cache line size (64 and 128 bytes), the I-cache size (1, 2, 4, and 8KB), the width of a bus between memory and the I-cache (32 and 64 bits), and the speed of processor core relative to memory (fast and slow). The D-cache (data cache) and I-cache have the same size and organization. The values of other SimpleScalar simulator parameters are shown in Table 1.

We assume that the AES decryption latency is 12 processor cycles for slow and 22 cycles for fast processor core, which are the speeds attainable with current optimized ASIC solutions [1]. Since a signature is always fetched first, signature decryption is finished before the corresponding instruction block is fetched, so the decryption latency is completely hidden in all evaluated system configurations. The address translation latency is one cycle for the SIGCED and SIGCEK techniques. For the SIGCEK and SIGCTK techniques, the S-cache has eight ways, random cache replacement policy, and twice as many entries as the corresponding I-cache. Note that an S-cache line contains only one signature of 16 bytes, whereas an I-cache line contains 64 or 128 bytes. Hence, the size of an I-cache with *n* cache lines is approximately two or four times larger than the size of an S-cache with *2n* entries.

Table 1. SimpleScalar Simulator parameters

Simulator parameter	Value
Branch predictor type	Bimodal
Branch predictor table size	128 entries, direct-mapped
Return address stack size	8 entries
Instruction decode bandwidth	1 instruction/cycle
Instruction issue bandwidth	1 instruction/cycle
Instruction commit bandwidth	1 instruction/cycle
Pipeline with in-order issue	True
I-cache/D-cache	4-way, first level only
I-TLB/D-TLB	32 entries, fully associative
Execution units	1 floating point, 1 integer
Memory fetch latency (first chunk/other chunks)	12/3 cycles for slow core, 24/6 cycles for fast core
Branch misprediction latency	2 cycles for slow core, 3 cycles for fast core
TLB latency	30 cycles for slow core, 60 cycles for fast core

The energy overhead is determined by comparing the total energy spent by a system with the Base configuration to the energy spent with signature verification. The total energy is calculated as a product of power dissipation and execution time. Power dissipation is estimated using a modified Sim-Panalyzer ARM simulator [19], which models the effects of internal and external switching and leakage. For Sim-Panalyzer parameters related to power, we use values from a provided template file. The operating frequency is 200MHz, and the supply voltage is 3.3V. The technology parameters correspond to the 0.18μm process. The AES decryption block is modeled as a static logic block with 10,000 gates. The S-cache is modeled as a regular cache structure.

We use benchmarks from several benchmark suites for embedded systems: MiBench [17], MediaBench [22], and Basicrypt [6]. Table 2 lists benchmarks, their descriptions, and the number of executed instructions, while Table 2 gives the number of I-cache misses per 1000 instructions.

Table 2. Benchmark description and characteristics

Benchmark	Description	Executed instructions in millions
blowfish_dec	Blowfish decryption	544.0
blowfish_enc	Blowfish encryption	544.0
cjpeg	JPEG compression	104.6
djpeg	JPEG decompression	23.4
ecdhb	Diffie-Hellman key exchange	122.5
ecdsignb	Digital signature generation	131.3
ecdsverb	Digital signature verification	171.9
ecelgdecb	El-Gamal encryption	92.4
ecelgencb	El-Gamal decryption	180.2
ispell	Spell checker	817.7
mpeg2_enc	MPEG2 compression	127.5
qsort	Quicksort	737.9
rijndael_dec	Rijndael decryption	307.9
rijndael_enc	Rijndael encryption	320.0
stringsearch	String search	3.7

Table 3 .Number of I-cache misses per 1000 instructions.

Benchmark	I-cache misses per 1000 instructions							
	Cache line 64B				Cache line 128B			
	1K	2K	4K	8K	1K	2K	4K	8K
blowfish_dec	22.2	5.6	0.1	0.0	13.7	3.8	0.8	0.0
blowfish_enc	22.2	4.6	0.1	0.0	12.9	3.8	0.8	0.0
cjpeg	6.2	1.6	0.3	0.1	6.6	1.7	0.3	0.1
djpeg	8.4	4.0	1.1	0.2	6.2	2.9	1.0	0.2
ecdhb	20.3	6.0	2.3	0.1	14.6	6.2	1.6	0.2
ecdsignb	15.9	4.6	1.7	0.1	17.3	4.8	1.2	0.1
ecdsverb	21.3	5.2	2.0	0.3	16.9	5.3	1.5	0.3
ecelgdecb	26.2	0.3	0.0	0.0	22.4	2.5	0.0	0.0
ecelgencb	23.4	3.2	1.1	0.1	18.7	4.4	0.8	0.1
ispell	61.7	51.1	21.7	2.9	40.4	35.7	20.9	3.5
mpeg2_enc	1.8	0.8	0.3	0.2	2.1	0.6	0.3	0.1
qsort	44.2	29.4	22.2	5.4	32.8	21.1	15.3	7.4
rijndael_dec	70.6	68.6	68.0	6.6	41.6	40.3	37.6	9.9
rijndael_enc	73.7	70.5	68.0	8.1	42.6	39.4	38.1	11.2
stringsearch	55.3	35.4	12.9	3.7	38.0	24.3	10.6	1.9

All benchmarks but *mpeg2encode* use the largest possible provided input, and *mpeg2encode* uses the provided test input. Since signature verification is done only at an I-cache miss, the benchmarks are selected so that most of them have a relatively high number of I-cache misses for at least some of the simulated cache sizes.

4. RESULTS

4.1 Performance Overhead

Figure 6 shows the performance overhead for a system with 64B I-cache lines, a 64-bit bus, and a slow processor core. The results indicate a low performance overhead for the SIGCED technique. With a 4K I-cache, the SIGCED technique increases CPI in the range 0.01-9.6%, with an average increase of 2.6%. Even with a very small 1K I-cache, the average CPI increase is 5.8%. The absolute CPI increase is very close to a linear function of the number of I-cache misses.

The SIGCED overhead can be reduced if signatures are kept in the S-cache, i.e., with the SIGCEK technique. With a 4K I-cache, the SIGCEK CPI increase is in the range 0.01-1.5%, with an average increase of 0.5%. With smaller I-caches, the SIGCEK CPI increase is in the range 0.3-7.4% (1K) and 0.3-5.7% (2K). A low number of I-cache misses with an 8K I-cache enables the SIGCEK to virtually remove the performance overhead of signature verification.

The SIGCTD technique always introduces more performance overhead than the SIGCED does, since signatures stored in the separate code section require an additional memory access. With a 4K I-cache, the SIGCTD increase is in the range 0.02-21%, and the average increase is 5.7%. This difference is more significant with small caches: The average CPI increase for the SIGCTD is 13% with a 1K I-cache and 8% with a 2K one.

The performance overhead can be reduced with an S-cache, i.e., with the SIGCTK technique. With a 4K I-cache, the SIGCTK CPI increase is in the range 0-1.2%, and the average increase is 0.2%.

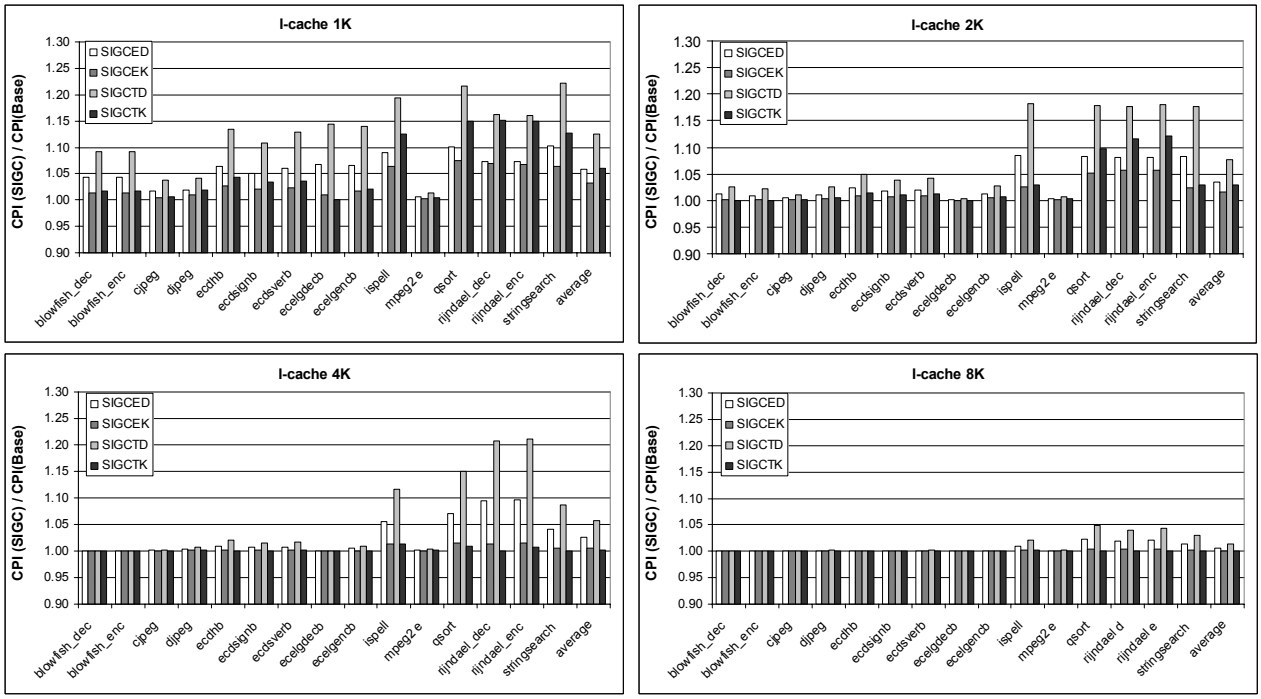

Figure 6. The ratio of CPI for the proposed techniques and the Base system.

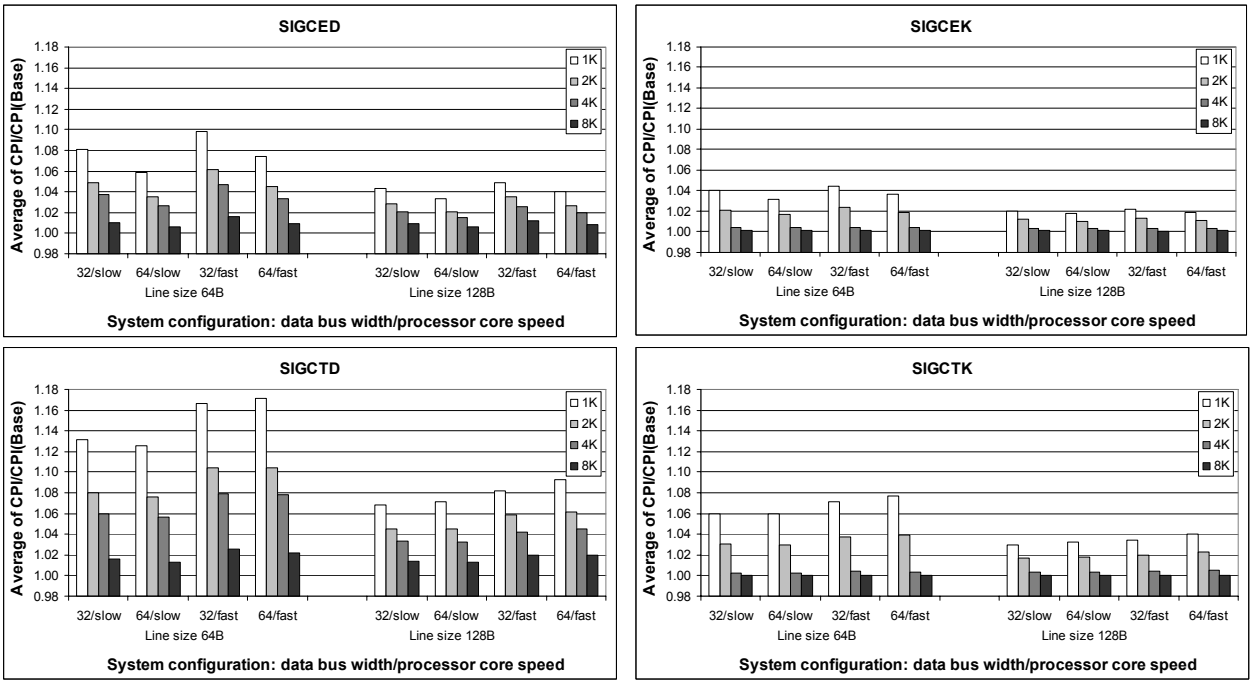

Figure 7. Sensitivity of performance overhead to architectural parameters.

Note that the overhead is less than with the SIGCEK technique; i.e., with this I-cache/S-cache size, the technique with a signature table performs better than does the technique with embedded signatures. This difference is due to the low number of S-cache misses with this configuration, on average less than 0.5 per 1000 instructions. With the SIGCTK, there is no additional delay if the signature is found in the S-cache, whereas with the SIGCEK there is always delay due to the address translation. However, with smaller caches the SIGCTK is worse than the SIGCEK: The average CPI increase is 6% with a 1K I-cache, and more than 10% for five benchmarks.

Figure 7 shows the sensitivity of the proposed techniques to processor core speed, memory bus width, and I-cache line size. The techniques with embedded signatures are less sensitive to the configuration parameters than the techniques with the signature table. For example, with a 2K I-cache the average CPI increase for SIGCED goes from 2% with 128B lines, 64-bit bus, and a slow core to 6% with 64B lines, 32-bit bus, and a fast core. The average CPI increase for the SIGCTD with the same I-cache varies from 4% to 10%. Techniques with an S-cache are also less sensitive than their counterparts without an S-cache, due to the lower CPI overhead. Note that performance overhead is higher with 64B than with 128B I-cache lines for all techniques, due to the larger number of I-cache misses (Table 2). The SIGCED technique remains an overall winner if the hardware budget does not allow for an S-cache. With a hardware budget insufficient for an I-cache increase but allowing for an S-cache, the SIGCEK technique has better overall performance than does the SIGCTK.

4.2 Energy Overhead

We evaluate the energy overhead of techniques with embedded signatures, since they have lower performance overhead. Figure 8 shows the energy of a system with the SIGCED and SIGCEK techniques normalized to the energy of the Base system, with 64B I-cache line, 64-bit memory bus, and slow core. With a 4K I-cache the SIGCED increases energy 0-27%, with an average increase of 8%; the SIGCEK technique reduces the average energy overhead to 3%. The increase in average power dissipation for the SIGCED is mainly due to the additional I/O activity when signatures are fetched from memory. Hence, both components of the SIGCED energy overhead are reduced with larger I-caches: The average energy increase is 16%, 10%, and 3%, for I-cache sizes of 1K, 2K, and 8K, respectively. This may not be the case with the SIGCEK technique. Whereas the S-cache may reduce the number of signature fetches and the corresponding performance and power penalties, it also increases the total die size and, consequently, the clock tree power dissipation. This is why the average SIGCEK energy increase with an 8K I-cache is larger than with a 4K one.

4.3 Memory Overhead

The memory overhead is an inherent characteristic of all proposed techniques, since instruction block signatures are added to the executable code sections. On average, the SIGCED technique with 16B signatures increases the size of the executable sections by 25.5% with 64B-instruction blocks, and by 14.3% with 128B-blocks. The SIGCTD technique does not require padding, so the executable section increase is 25% with 64B cache lines and 12.5% with 128B cache lines. An executable file typically includes non-executable code sections, so the proposed techniques add even less memory overhead to complete executable files: less than 7% with 64B cache lines, and less than 4% with 128B lines.

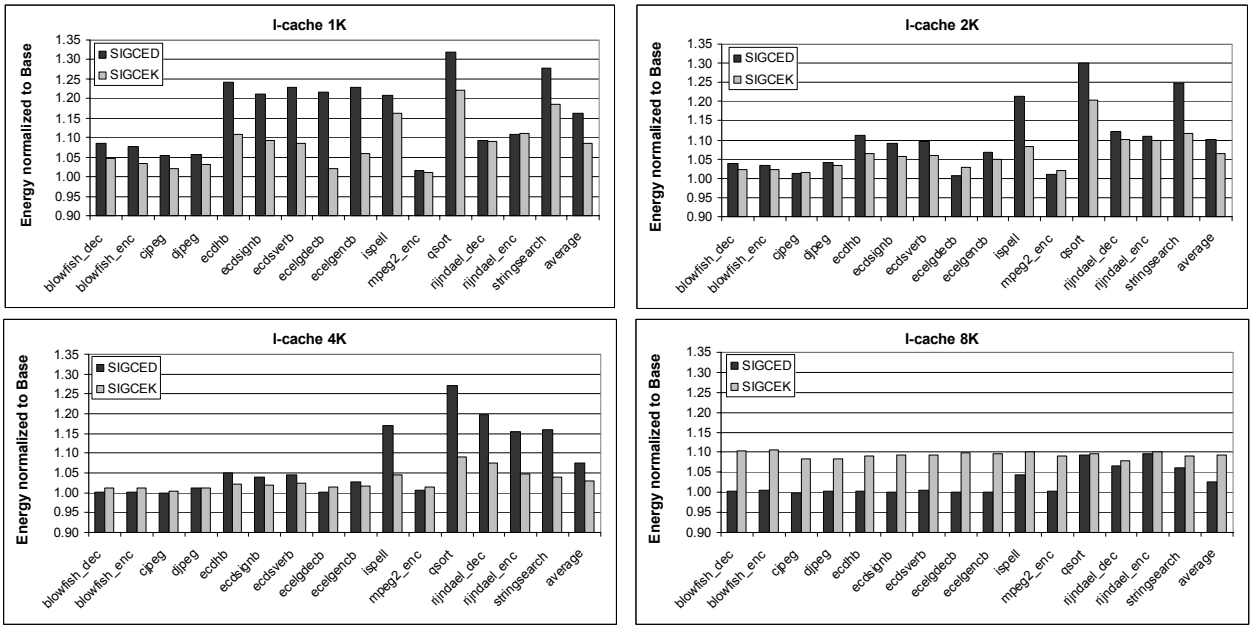

Figure 8. System energy normalized to the Base case.

5. RELATED WORK

A broad spectrum of techniques has been proposed to counter code injection attacks. These techniques can be classified into static software based, dynamic software based, and techniques that include hardware support. Static software based techniques attempt to find possible security vulnerabilities in the code, so they can be corrected before the code is released. Completely automated tools for detection of security-related flaws must choose between precise but not scalable analysis and lightweight analysis that may produce a lot of false positives and false negatives [41]. The need for precise automated analysis can be alleviated if programmers add specially formulated comments about program constraints [14], but adding annotations can be as error prone as programming itself and puts additional burden on programmers.

Dynamic software techniques aim to prevent successful code injection attacks or to significantly reduce the attackers' chances for success. These techniques can be further classified into four groups. One group encompasses techniques that augment the code with various run-time checks [11, 16, 24]. Another group comprises techniques that monitor different aspects of program behavior, such as sequences of system calls or values of performance monitoring registers [32, 34]. The third approach is obfuscation: This group includes segment addresses and jump addresses, or the complete code can be scrambled, making it difficult for an attacker to succeed [5, 7, 10]. Finally, the fourth group includes "safe dialects" of language C, which restrict the use of unsafe constructs, perform static analysis and runtime checks, and use garbage collection or region-base memory management [21]. Dynamic software techniques often require recompilation, so they are not readily applicable to legacy code. Moreover, since these techniques increase the code size and the number of executed instructions, they may incur significant performance and energy overhead.

Hardware-based defense techniques use architectural support to counter injected code. These techniques promise lower overhead in performance and energy than do solely software solutions. They also reduce overall defense cost: An architectural feature protects all programs executed by the processor, whereas most software-based techniques have to be applied to each program separately. However, hardware-based techniques are mainly attack-specific. For example, several researchers propose architectural support against stack smashing attacks, either by saving return addresses on a separate hardware stack [23, 33, 43, 45], or by interpreting instructions in hardware so that return addresses are saved and restored from a "shadow" stack [9]. Various obfuscation techniques can also benefit from hardware support: from using special instructions to load/store encrypted code pointers [35, 38], to transforming instruction blocks according to the encrypted hash values of transformation invariants [20], to complete code encryption [18]. Another approach is to tag untrustworthy data that cannot be used for control transfers [12, 37]. For applications with intensive I/O tagging may have a significant performance and power overhead.

The code integrity in run-time can be successfully protected if all instruction blocks are signed with a cryptographically secure signature. We did preliminary research on protection of basic blocks and cache blocks using signatures [28, 29]. The results of this research indicated that basic block signatures might add a significant overhead. Cache block signatures were evaluated with a less detailed performance simulator and without the S-cache. Independently, Kirovski et al. also propose to sign all cache blocks and to verify signatures in run-time [15]. This approach generates the signature for an instruction block using a chained Rijndael cipher: The instruction block is divided into sub-blocks, so that each cipher stage encrypts the result of XOR of a sub-block and the output from the previous stage. Such encoding might be more cryptographically secure, but it introduces more power and performance overhead. The authors did not consider

the S-cache, and tried to reduce the impact of signature mechanism by code transformation.

Signatures of instruction blocks of various granularity are also frequently used in fault-tolerant computing [26, 36, 42]. Unlike these techniques, our approach does not require a dedicated watchdog processor, and focuses rather on seamless integration on verification mechanism into existing processor architecture. Moreover, the signatures in our mechanism are also protected from read attacks.

The ultimate protection of code integrity can be achieved if all instructions are encrypted, as in the eXecute Only Memory (XOM) framework [25]. However, complete code encryption by a cryptographically strong technique such as AES or DES considerably slows down execution. A fast on-time pad encryption has been proposed for XOM, where pairs of instructions are XORed with the corresponding encrypted address [44]. This approach might be vulnerable to attacks where an attacker is able to correctly guess the instructions in an instruction pair, which than can be replaced by a malicious pair.

6. CONCLUSION

This paper presents a hardware mechanism that provides complete run-time code integrity and evaluates four different implementations of that mechanism in terms of additional performance and energy overhead. The technique with embedded signatures that are discarded after verification has relatively low hardware complexity and a very low overhead across all considered configurations; this overhead can be further reduced with an additional processor resource, a signature cache. The techniques with signatures in a separate code section have higher overhead and slightly simpler implementation than the techniques with embedded signatures.

Low overhead, protection from the whole class of code injection attacks, and applicability to already-compiled code make the proposed techniques a better choice for embedded systems than run-time checking techniques implemented solely in software. The signature verification mechanism also detects unintentional code changes that may happen in error-prone environments.

In future work, performance overhead for all signature verification techniques might be reduced with prefetching of code and signatures. The signature verification mechanism can be expanded to provide defense from other types of attacks, such as replay attacks and return-into-libc.

REFERENCES

[1] "Enhanced Aes (Rijndael) IP Core," <http://www.asics.ws> (Available December 2004).

[2] "Intel Xscale® Core Developer's Manual," <http://www.intel.com/design/intelxscale/> (Available December 2004).

[3] Ahmad, D., "The Rising Threat of Vulnerabilities Due to Integer Errors," *IEEE Security & Privacy*, 1, 4 (July-August 2003), 77-82.

[4] Austin, T., Larson, E., and Ernst, D., "Simplescalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, 35, 2 (February 2002), 59-67.

[5] Bhatkar, S., DuVarney, D. C., and Sekar, R., "Address Obfuscation: An Approach to Combat Buffer Overflows, Format-String Attacks, and More," in *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, USA, 2003, 105-120.

[6] Branovic, I., Giorgi, R., and Martinelli, E., "A Workload Characterization of Elliptic Curve Cryptography Methods in Embedded Environments," *ACM SIGARCH Computer Architecture News*, 32, 3 (June 2004), 27-34.

[7] Busser, P., "Memory Protection with Pax and the Stack Smashing Protector: Breaking out Peace," *Linux Magazine*, 40(March 2004), 36-39.

[8] Conover, M., "w00w00 on Heap Overflows," <http://www.w00w00.org/files/articles/heaptut.txt> (Available January 2005).

[9] Corliss, M., Lewis, E. C., and Roth, A., "Using DISE to Protect Return Addresses from Attack," in *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, Boston, MA, USA, 2004, 61-68.

[10] Cowan, C., Beattie, S., Johansen, J., and Wagle, P., "Pointguard™: Protecting Pointers from Buffer Overflow Vulnerabilities," in *Proceedings of the 12th USENIX Security Symposium*, Washington, DC, USA, 2003, 91-104.

[11] Cowan, C., Pu, C., Maier, D., Walpole, J., Bakke, P., Beattie, S., Grier, A., Wagle, P., Zhang, Q., and Hinton, H., "Stackguard: Automatic Adaptive Detection and Prevention of Buffer Overflow Attacks," in *Proceedings of the 7th USENIX Security Conference*, San Antonio, TX, USA, 1998, 63-78.

[12] Crandall, J. R. and Chong, F. T., "Minos: Control Data Attack Prevention Orthogonal to Memory Model," in *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Portland, OR, USA, 2004, 221-232.

[13] Dobrovitski, I., "Exploit for Cvs Double free() for Linux pserver," <http://seclists.org/lists/bugtraq/2003/Feb/0042.html> (Available January 2005).

[14] Dor, N., Rodeh, M., and Sagiv, M., "CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, San Diego, CA, USA, 2003, 155-167.

[15] Drinic, M. and Kirovski, D., "A Hardware-Software Platform for Intrusion Prevention," in *Proceedings of the 37th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Portland, OR, USA, 2004, 233-242.

[16] Fetzer, C. and Xiao, Z., "Detecting Heap Smashing Attacks through Fault Containment Wrappers," in *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, New Orleans, LA, USA, 2001, 80-89.

[17] Guthaus, M. R., Ringenberg, J. S., Ernst, D., Austin, T. M., Mudge, T., and Brown, R. B., "MiBench: A Free, Commercially Representative Embedded Benchmark Suite," in *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, Austin, TX, USA, 2001.

[18] Kc, G. S., Keromytis, A. D., and Prevelakis, V., "Countering Code-Injection Attacks with Instruction-Set Randomization," in *Proceedings of the 10th ACM Conference on Computer*

and Communication Security, Washington, DC, USA, 2003, 272-280.

[19] Kim, N., Kgil, T., Bertacco, V., Austin, T., and Mudge, T., "Microarchitectural Power Modeling Techniques for Deep Sub-Micron Microprocessors," in *Proceedings of the International Symposium on Low Power Electronics and Design (ISLPED)*, Newport Beach, CA, USA, 2004, 212-217.

[20] Kirovski, D., Drinic, M., and Potkonjak, M., "Enabling Trusted Software Integrity," in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, San Jose, CA, USA, 2002, 108-120.

[21] Larus, J. R., Ball, T., Das, M., DeLine, R., Fähndrich, M., Pincus, J., Rajamani, S. K., and Venkatapathy, R., "Righting Software," *IEEE Software*, 21, 3 (May-June 2004), 92-100.

[22] Lee, C., Potkonjak, M., and Mangione-Smith, W. H., "MediaBench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," *IEEE Micro*, 30, 1 (December 1997), 330-335.

[23] Lee, R. B., Karig, D. K., McGregor, J. P., and Shi, Z., "Enlisting Hardware Architecture to Thwart Malicious Code Injection," in *Proceedings of the Security in Pervasive Computing*, Boppard, Germany, 2003, 237-252.

[24] Lhee, K.-s. and Chapin, S. J., "Type-Assisted Dynamic Buffer Overflow Detection," in *Proceedings of the 11th USENIX Security Symposium*, San Francisco, CA, USA, 2002, 81-88.

[25] Lie, D., Thekkath, C., Mitchell, M., Lincolny, P., Boneh, D., Mitchell, J., and Horowitz, M., "Architectural Support for Copy and Tamper Resistant Software," in *Proceedings of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems*, Cambridge, MA, USA, 2000, 168-177.

[26] Mahmood, A. and McCluskey, E. J., "Concurrent Error Detection Using Watchdog Processors-a Survey," *IEEE Transactions on Computers*, 37, 2 (February 1988), 160-174.

[27] Milenkovic, M., "Architectures for Run-Time Verification of Code Integrity," Ph.D. Thesis, Electrical and Computer Engineering Department, University of Alabama in Huntsville, 2005.

[28] Milenkovic, M., Milenkovic, A., and Jovanov, E., "A Framework for Trusted Instruction Execution Via Basic Block Signature Verification," in *Proceedings of the 42nd Annual ACM Southeast Conference*, Huntsville, AL, USA, 2004, 191-196.

[29] Milenkovic, M., Milenkovic, A., and Jovanov, E., "Using Instruction Block Signatures to Counter Code Injection Attacks," in *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, Boston, MA, USA, 2004, 104-113.

[30] Newsham, T., "Format String Attacks," September 2000, <http://www.securityfocus.com/guest/3342> (Available January 2004).

[31] One, A., "Smashing the Stack for Fun and Profit," *Phrack Magazine*, 7, 49 (November 1996).

[32] Oppenheimer, D. L. and Martonosi, M. R., "Performance Signatures: A Mechanism for Intrusion Detection," in *Proceedings of the 1997 IEEE Information Survivability Workshop*, San Diego, CA, USA, 1997.

[33] Ozdoganoglu, H., Brodley, C. E., Vijaykumar, T. N., Kuperman, B. A., and Jalote, A., "SmashGuard: A Hardware Solution to Prevent Security Attacks on the Function Return Address," Purdue University, TR-ECE 03-13, November 22, 2003.

[34] Sekar, R., Bendre, M., Dhurjati, D., and Bollineni, P., "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors," in *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 2001, 144-155.

[35] Shao, Z., Zhuge, Q., He, Y., and Sha, E. H.-M., "Defending Embedded Systems against Buffer Overflow Via Hardware/Software," in *Proceedings of the 19th Annual Computer Security Applications Conference (ACSAC 2003)*, Las Vegas, NV, USA, 2003, 352-363.

[36] Shen, J. P. and Schuette, M. A., "On-Line Self-Monitoring Using Signatured Instruction Streams," in *Proceedings of the 1983 IEEE International Test Conference*, Philadelphia, PA, USA, 1983, 275-282.

[37] Suh, G. E., Lee, J. W., and Devadas, S., "Secure Program Execution Via Dynamic Information Flow Tracking," in *Proceedings of the 11th Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, Boston, MA, USA, 2004, 85-96.

[38] Tuck, N., Calder, B., and Varghese, G., "Hardware and Binary Modification Support for Code Pointer Protection from Buffer Overflow," in *Proceedings of the 37th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO)*, Portland, OR, USA, 2004, 209-220.

[39] US-CERT, "Cert/Cc Statistics," <http://www.cert.org/stats/> (Available December 2003).

[40] US-CERT, "Cyber Security Bulletin Sb04-231," <http://www.us-cert.gov/cas/bulletins/SB04-231.html> (Available November 2004).

[41] Wagner, D., Foster, J. S., Brewer, E. A., and Aiken, A., "A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities," in *Proceedings of the Network and Distributed System Security Symposium (NDCS)*, San Diego, CA, USA, 2000.

[42] Wilken, K. and Shen, J. P., "Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 9, 6 (June 1990), 629-641.

[43] Xu, J., Kalbarczyk, Z., Patel, S., and Iyer, R. K., "Architecture Support for Defending against Buffer Overflow Attacks," in *Proceedings of the Workshop on Evaluating and Architecting System dependability (EASY)*, San Jose, CA, USA, 2002.

[44] Yang, J., Zhang, Y., and L.Gao, "Fast Secure Processor for Inhibiting Software Piracy and Tampering," in *Proceedings of the 36th International Symposium on Microarchitecture*, San Diego, CA, USA, 2003, 351-360.

[45] Ye, D. and Kaeli, D., "A Reliable Return Address Stack: Microarchitectural Features to Defeat Stack Smashing," in *Proceedings of the Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, Boston, MA, USA, 2004, 69-76.