An Efficient Runtime Instruction Block Verification for Secure Embedded Systems

Aleksandar Milenkovic, Milena Milenkovic, and Emil Jovanov

Abstract—**Embedded system designers face a unique set of challenges in making their systems more secure, as these systems often have stringent resource constraints or must operate in harsh or physically insecure environments. One of the security issues that have recently drawn attention is software integrity, which ensures that the programs in the system have not been changed either by an accident or an attack. In this paper we propose an efficient hardware mechanism for runtime verification of software integrity using encrypted instruction block signatures. We introduce several variations of the basic mechanism, and give details of three techniques that are most suitable for embedded systems. Performance evaluation using selected MiBench, Mediabench, and Basicrypt benchmarks indicates that the considered techniques impose a relatively small performance overhead. The best overall technique has performance overhead in the range 0-8%, when protecting 128-byte instruction blocks with 16-byte signatures. With 64-byte instruction blocks, the overhead is in the range 0-15%; the average overhead with 8 KB cache is 1%. With additional investment in a signature cache, this overhead can be almost completely eliminated.**

Index Terms—**Computer architecture, embedded systems, secure computing, processor design, performance evaluation, security attacks, decryption.**

I. Introduction

"THE art of war teaches us to rely not on the likelihood of the enemy's not coming, but on our own readiness to receive him; not on the chance of his not attacking, but rather on the fact that we have made our position unassailable."
The Art of War by Sun Tzu

Embedded systems have become ubiquitous in modern society, finding their place in a broad range of applications, from military to health monitoring. Economic and technology trends will further increase our reliance on the embedded systems as we move toward new applications featuring smart environments, built on highly interconnected and deeply embedded computing systems. These trends further underscore the utmost importance of embedded system security. Failing to resist to attacks can incur significant direct costs as well as costs in lost revenue opportunities.

A very large group of malicious attacks on applications running on general-purpose processors comprises of different techniques that impair the software integrity, by injecting and then executing the malicious code instead of regularly installed programs. The most widely known type of such attacks is so-called stack smashing. In this attack an attacker overflows a buffer stored on the stack with a malicious code sequence and replaces a valid return address with the malicious code address [1]. Various other examples of attacks exist, such as heap overflow and format string attacks [2].

Applications targeting embedded systems may suffer from the same vulnerabilities as applications running on general-purpose platforms. For example, one recent Cyber Security Bulletin from United States Computer Emergency Readiness Team (US-CERT) reports multiple buffer overflow vulnerabilities in a Bluetooth connectivity program for Personal Digital Assistants (PDAs) [3]. Another US-CERT Cyber Security Bulletin indicates an emerging trend of mobile phone viruses [4]. As the communication and computation capabilities of smart phones, PDAs, and other embedded systems continue to grow, so will grow the number of malicious attacks trying to exploit code vulnerabilities.

The multitude of code injection attacks on general-purpose processors prompted a large number of predominantly software-based counter-measures. The software-based techniques can be classified into static techniques, which detect security defects in compile time, and dynamic techniques, which augment the code to detect attacks in runtime. Several hardware-supported techniques have also been proposed recently. For example, hardware protection from buffer overflow has already found its way in main stream general-purpose processors, AMD's Athlon-64 and Intel's Itanium [5].

An acceptable performance overhead of a defense technique in a high-speed general-purpose processor may translate into an undesirable energy and performance overhead in an embedded system. Whereas static software techniques can be readily applied to embedded systems software, the inherent overhead of the most of dynamic techniques makes them a poor choice for embedded systems. Hardware techniques have the potential to improve resilience to code injection attacks with less power and performance overhead. Hardware design of embedded systems is not encumbered by legacy issues and complexity of general-

Manuscript received December 15, 2004.

A. Milenkovic is with the Electrical and Computer Engineering Department, University of Alabama in Huntsville, Huntsville, AL 35899 USA (phone: 256-824-6830; fax: 256-824-6803; e-mail: milenka@ece.uah.edu).

M. Milenkovic is with IBM, Austin, TX 78758 USA (e-mail: milena@computer.org).

E. Jovanov is with the Electrical and Computer Engineering Department, University of Alabama in Huntsville, Huntsville, AL 35899 USA (e-mail: jovanov@ece.uah.edu).

purpose platforms, so the hardware support for different security layers can be implemented relatively easily. For example, ARM's TrustZone [6] secures data on a chip by integrating several security features with the processor core. However, most of the existing hardware techniques are attack-specific. 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 novel hardware-supported techniques to ensure software integrity in embedded systems. All proposed techniques employ a common mechanism: instruction blocks are signed using secret hardware keys during secure installation process, and signatures are stored together with the code. During program execution, signatures are recalculated from instructions and compared with the stored values. If two values do not match, the software integrity of the application does not hold any more, and the application is terminated by the operating system. We classify all such techniques and evaluate three that look the most promising for embedded systems in terms of added complexity, performance overhead, and power consumption: SIGCED, SIGCEK, and SIGCEV. With these techniques the protected instruction block is of the size of a cache block, the block signatures are embedded in the code, and a signature is verified only at an instruction cache (I-cache) miss. None of the three considered techniques require compiler support. The SIGCED technique discards signatures after successful verification. Instruction addresses are translated in such a way that both the processor and the I-cache see instructions at the addresses as they would appear without embedded signatures. With the signature size of 16 B, this technique has 0-8% performance overhead when the protected block size is 128 B, and 0-15% when the protected block size is 64 B, depending on the benchmark, the instruction cache size, the memory bus width, and the speed of the processor core relative to the memory speed. The performance and power overhead may be reduced if signatures are kept in a signature cache (S-cache), as in the SIGCEK technique. An S-cache that is 25% of the size of the corresponding I-cache may reduce the signature verification overhead for up to 90%. The third technique, the SIGCEV, also discards signatures, but instructions are stored in the I-cache without address translation. With the same cache line address aligning, the SIGCEV I-cache is actually smaller than the original I-cache. The code is mapped into the I-cache in a different way, so for some benchmarks the performance with the SIGCEV even improves, whereas for others the performance significantly deteriorates due to more I-cache misses. Each of the three techniques is most suitable for a particular hardware budget: if the hardware budget is very low, the smaller SIGCEV I-cache might be an acceptable solution. Otherwise, the consistent small overhead of the SIGCED technique makes it a perfect candidate for efficient runtime verification of software integrity. With larger I-caches the performance overhead of this technique is close to 0%. Finally, if a hardware budget is flexible but does not allow doubling the I-cache size, extra area on the chip is well invested into the S-cache of the SIGCEK technique.

This paper is organized as follows. Section II describes the proposed architectures for instruction block verification. Section III describes the experimental methodology and Section IV discusses the results of the performance analysis. Section V describes the related work and the last section concludes the paper.

II. Architectures for Instruction Block Verification

In this section we first introduce the basic mechanism common to all techniques for instruction block verification discussed in this paper. We then give a taxonomy of different techniques and discuss pros and cons. Finally, we present a detailed description of three techniques evaluated in this paper.

A. Basic Mechanism For Runtime Instruction Block Verification

All proposed architectures for instruction block verification have the same basic mechanism (Fig. 1) and require minimal or no compiler support.

Secure Installation. During a secure installation process, signatures are calculated for each instruction block and added to the binary program. The signature of an instruction block should be relatively fast to calculate and verify. On the other hand, it should be very hard, preferably impossible, for an attacker to generate a correct signature for the code that he/she wants to inject. Consequently, we decided to use an extended version of a Multiple Input Shift Register (MISR) and Advance Encryption Standard (AES) [7] to generate and verify signatures. MISRs are frequently used in VLSI testing, since they compress an array of data under test into one signature, which is then compared to the signature of a known correctly functioning component. A signature is obtained in the following way: all instructions in the instruction block pass through a MISR. A MISR is essentially an array of D flip-flops with linear feedback coefficients (Fig. 2): a new value of MISR is function of the current value and value of incoming instructions. After each instruction block, the MISR is initialized to a predefined start value, e.g., $K_0K_1K_2K_3$ in Fig. 2. Linear feedback connections and the start value are determined by a secret processor key hidden in hardware. 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. However, in our approach the signatures are further encrypted using AES, which is proved to be secure. The AES key is also stored in hardware and thus hidden from attacker.

Program Execution. Signatures are verified in parallel with program execution using a dedicated hardware resource Instruction Block Signature Verification Unit (IBSVU, Fig. 3). Since the I-cache is a read-only resource, instruction block signatures are verified only on I-cache misses. Fetched instructions pass through a MISR register with the linear coefficients used during secure installation, while concurrently the corresponding signature fetched from memory is decrypted. 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, so a trap to the operating system is asserted. The operating system than kills the process whose code integrity cannot be guaranteed and possibly audits the event.

An embedded system might be designed to run only in the protected mode where all executing instruction blocks must be signed, as described above. However, some applications do not need instruction block protection. For example, some components of the operating system may not accept external inputs from untrustworthy channels and thus are not in danger from code injection attacks. Such programs may be installed without signatures and executed in the unprotected mode. The information about required execution mode is added to the program header.

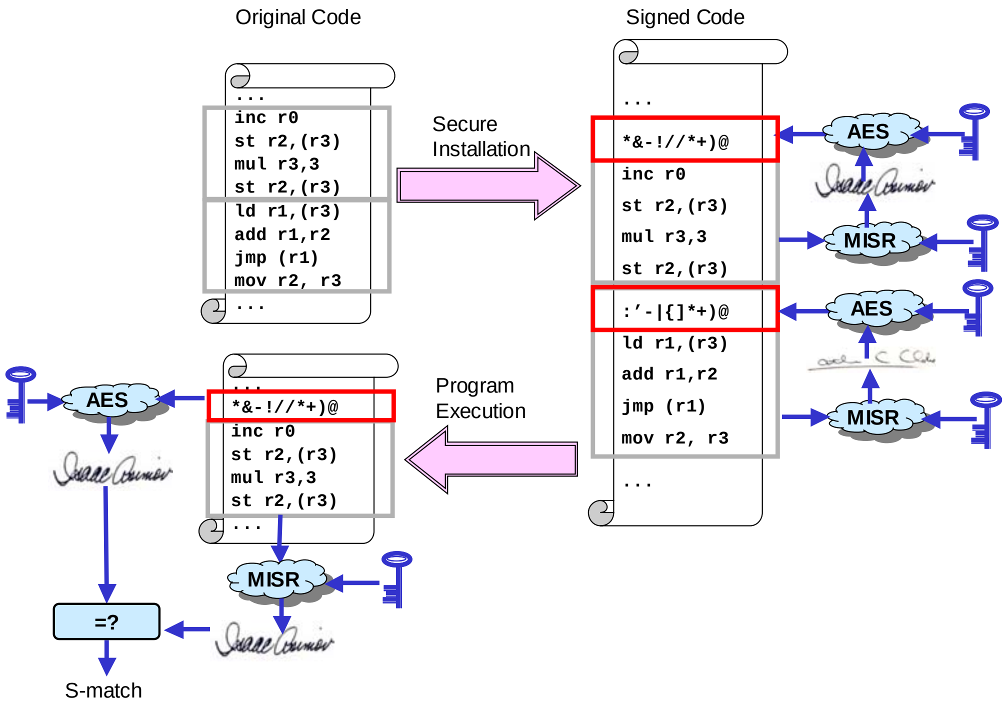

Fig. 1 Mechanism for trusted instruction execution

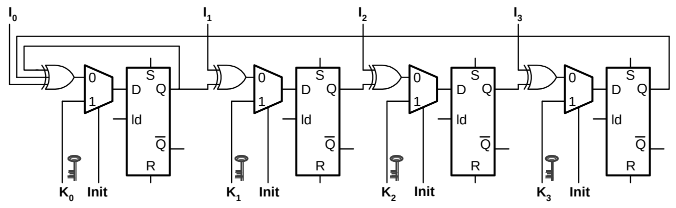

Fig. 2 An implementation of a 4-bit MISR

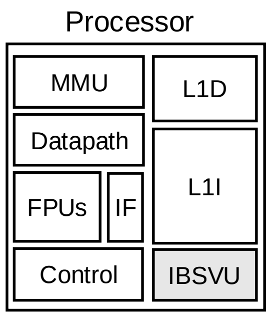

Fig. 3 Processor modifications

B. Taxonomy of Techniques For Runtime Instruction Block Verification

Instruction block verification techniques can be classified according to the following criteria:

- type of protected instruction blocks,
- signature placement,
- signature handling after verification,
- signature visibility to the I-cache.

The taxonomy of instruction block verification techniques is given in Fig. 4. The name of a verification technique starts with SIG, and the rest of the name specifies the categories to which the technique belongs. For example, the SIGCED technique protects instruction blocks of the size equal to the size of an I-cache line (C) with signatures embedded in the code (E) and disposed after verification (D).

A protected block can be of variable or fixed size. With variable-size blocks, one signature protects a logical code unit such as a basic block or an instruction stream (dynamic basic block). With fixed-size blocks, one signature protects a physical code unit of the size equal to the size of one or more I-cache lines.

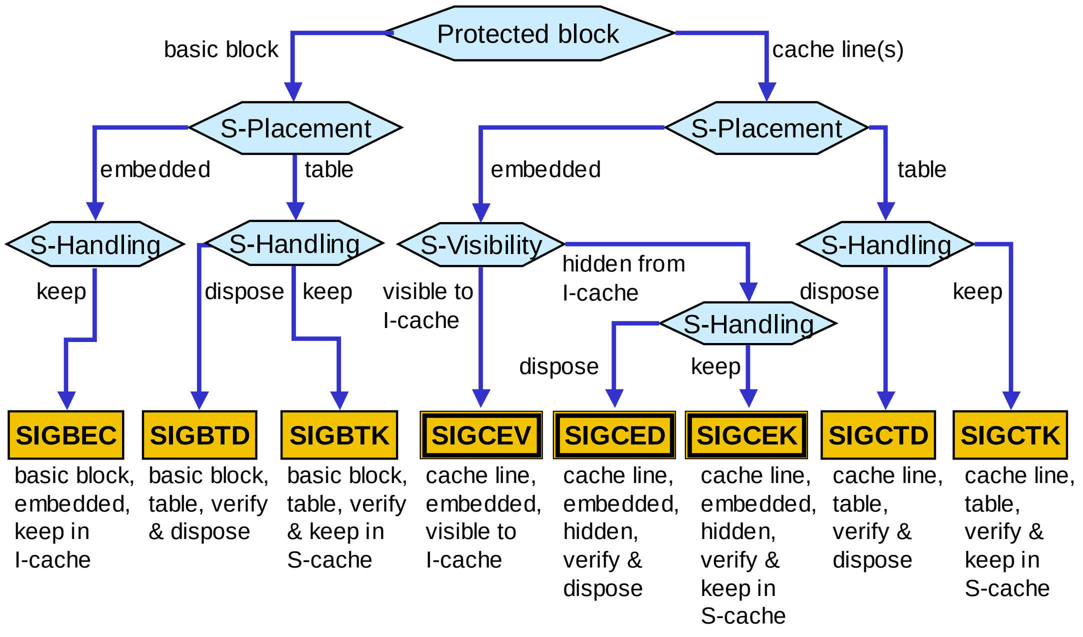
Protected block
basic block
cache line(s)
S-Placement
S-Placement
embedded
table
embedded
table
S-Handling
S-Handling
S-Visibility
S-Handling
keep
dispose
keep
visible to I-cache
hidden from I-cache
dispose
keep
S-Handling
dispose
keep
SIGBEC
SIGBTD
SIGBTK
SIGCEV
SIGCED
SIGCEK
SIGCTD
SIGCTK
basic block, embedded, keep in I-cache
basic block, table, verify & dispose
basic block, table, verify & keep in S-cache
cache line, embedded, visible to I-cache
cache line, embedded, hidden, verify & dispose
cache line, embedded, hidden, verify & keep in S-cache
cache line, table, verify & dispose
cache line, table, verify & keep in S-cache

Fig. 4 Taxonomy of proposed instruction block verification techniques

Verification techniques can be further classified depending on the placement of signatures in a binary file. A signature can be embedded in the code, i.e., placed before or after the instruction block it protects. Another option is to store all signatures in a separate table, i.e., a separate code section.

A basic block protection technique with embedded signatures must keep signatures together with the instructions in the I-cache, since embedded signatures cannot be extracted from the code in the fetch stage without decoding [8]. The name of this technique in our taxonomy is SIGBEC. With the SIGBEC technique, the instruction decoder must be able to tell the difference between a signature and a regular instruction. This can be achieved by reserving one instruction bit for the signature flag, or by using a special opcode that indicates to the decoder that instruction words immediately following it represent a signature.

Basic block protection techniques with signatures stored in the separate code section work in the following way. When the instruction decoder detects the end of a basic block that caused at least one cache miss, the signature of that block must be fetched from the signature code section, decrypted, and compared to the calculated signature. These techniques can be classified depending on whether a decrypted signature is kept in a dedicated S-cache after verification (SIGBTK) [9], or it is disposed of (SIGBTD). With the SIGBTD technique, there is no overhead for signature fetch from memory if a signature that needs to be verified is found in the S-cache.

In all techniques with protected basic blocks, signatures can be verified after decode stage, i.e., when the last instruction in a basic block has been decoded. These techniques require compiler support, since disassembling in general case cannot extract the basic block list from the executable code with 100% accuracy [10]. However, the required support is relatively simple, since the program compilation process only needs to generate a list of all basic blocks in the code and to append it to the executable. Embedded signatures are converted to no-ops in the decode stage, so they are visible only to the dedicated signature verification unit and not to the rest of the processor core. However, instruction addresses will change due to embedded signatures, so the installation process must recalculate all target addresses. Hence, the list of basic block must also include target addresses. Further description and evaluation of these techniques can be found in [8], [9].

Compiler support is not necessary for techniques protecting instruction blocks of a fixed size. Moreover, the signatures can be verified in parallel with the fetch stage, since the exact placement of signatures and protected blocks is known in advance. Embedded cache line signatures are not visible to the processor, i.e., the processor is aware only of the executable code. This invisibility is achieved with the use of a relatively simple address translation, so that the processor "sees" instruction addresses as if there were no embedded signatures. The address translation can be done before or after the instructions are stored in the I-cache, that is, the signatures can be hidden from the I-cache or visible to it (SIGCEV in our taxonomy). If signatures are hidden from the I-cache, they can be disposed of after verification (SIGCED) or kept in the S-cache (SIGCEK). As in the previously described techniques, cache line signatures placed in a separate code section can be discarded after verification (SIGCTD) or kept in the S-cache (SIGCTK).

Table 1 illustrates the most important pros and cons of the proposed techniques. Relevant parameters include the need for compiler support; hardware complexity, i.e., the estimated area on the chip required by a particular technique; the projected performance overhead, based on the delays that a technique introduces to program execution; and the requirement to change the instruction set architecture (ISA).

Table 1 Pros and cons of different techniques

	Compiler support	**Hardware complexity**	**Projected performance overhead**	**ISA change**
SIGBEC	Yes	Low	Medium	Yes
SIGBTD	Yes	Low	Medium	No
SIGBTK	Yes	Medium	Low	No
SIGCEV	No; may be used	Low	Low to medium	No
SIGCED	No; may be used	Low	Low to medium	No
SIGCEK	No; may be used	Medium	Low	No
SIGCTD	No	Low	Low to medium	No
SIGCTK	No	Medium	Low	No

As explained before, the techniques protecting the basic blocks, SIGBEC, SIGBTD, and SIGBTK, require compiler support, while the techniques protecting cache lines, SIGC, do not. However, the SIGCE techniques (protected cache lines with embedded signatures) may benefit from compiler support. The branch target addresses change due to embedded signatures, so either a compiler recalculates all target addresses, or address translation is done in hardware. In the rest of the paper we assume that SIGCE techniques use hardware address translation.

All proposed techniques require a relatively simple processor modification: a dedicated processor resource for signature verification, the IBSVU (Fig. 3). The IBSVU encompasses registers for buffering instructions and signatures, support for AES decryption, MISR, and control logic. Techniques that keep signatures in a signature cache require additional on-chip area, so they are marked as having Medium hardware complexity in Table 1: SIGBTK, SIGCEK, and SIGCTK.

The overhead of fetching a signature from the memory and its decryption is avoided if the signature is found in the S-cache, so the techniques with the S-cache have a low projected performance overhead. The SIGBEC and SIGBTD techniques have potentially higher performance overhead than the corresponding SIGC techniques. With the SIGBEC embedded basic block signatures may reduce the number of cache hits, leading to a medium performance overhead [8]. With the SIGCTD the access function of a table of instruction block signatures in memory is relatively simple, whereas with the SIGBTD a more complex hash function must be used to access a table of basic block signatures, thus adding additional latency.

The advantage of the basic block protection techniques is that only instructions that will be executed are verified, whereas only a portion of instructions in a cache line might be really needed. All techniques but one, the SIGBEC, do not require the change of the processor instruction set.

The techniques with protected cache line blocks and embedded signatures (SIGCE) are the most suitable for embedded systems. These techniques do not require compiler support and use simpler algorithms than techniques with signatures stored in a table, so they are simpler to implement and less expensive.

C. Details of SIGCE techniques

In this section we will explain details of three SIGCE techniques. These techniques are:

- SIGCED – signatures are invisible to the I-cache and discarded after verification;
- SIGCEK – signatures are invisible to the I-cache and kept in the S-cache;
- SIGCEV – signatures are visible to the I-cache.

We assume that all three techniques do not use compiler support, i.e., the original binary is modified during the secure installation process only by inserting signatures and necessary padding. If the last instruction block is shorter than the cache line, it is padded by instructions that do not change the state of the processor. If the code with embedded signatures is larger than a page size, it must be page aligned, so additional padding is necessary for each page but the last.

1) SIGCED

The flow of the instruction fetch phase is depicted in Fig. 5. The value of the next 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 virtually tagged. This is a frequent case in embedded processor caches, for example in Intel's Xscale processor [11]. In the case of a cache hit, the instruction is fetched from the I-cache and there is no need for instruction verification. In the case of an I-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. If the padding is not necessary, i.e., one memory page can be completely filled with the protected instruction blocks and corresponding signatures, the true virtual address *tPCtemp* can be calculated as in (1). The value *SigSize* is the signature size, *BlockSize* is the protected block size, and *TextBase* is the starting address of the text segment for a given program.

$$tPCtemp = PC + SigSize \cdot \left(\frac{PC\text{-}TextBase}{BlockSize} + 1 \right) \qquad (1)$$

The size of the padding *PagePad* is given in (2), with *PageSize* denoting the memory page size. The final true address *tPC* can be calculated as in (3).

$$PagePad = PageSize \bmod (BlockSize + SigSize) \qquad (2)$$

$$tPC = tPCtemp + \frac{tPCtemp\text{-}TextBase}{PageSize\text{-}\ PagePad} \cdot PagePad \qquad (3)$$

For example, consider a case where the I-cache line is 128 B, the signature size is 16 B, the page size is 4096 B, the

TextBase address is 131072, and the value of the PC of the instruction to be fetched as seen by the processor is 135200. In the original code without signatures, the size of a page is equal to the size of 32 instruction blocks. In the signed code, the size of a protected block together with its signature is 144 B. Hence, 28 signed blocks can fit in one page, filling 4032 out of 4096 B. Since one instruction block cannot be split between two pages, the code must be padded so that the remaining 64 B in a page are unused. All instruction blocks must have the same size, so if the last instruction block in a binary is shorter than the I-cache block, it is padded with randomly chosen instructions that do not change the state of the processor.

When a correct virtual address is calculated, the translation look-aside buffer (TLB) is accessed for virtual to physical address translation. In all considered SIGCE techniques a signature is inserted into the code just before the corresponding protected instruction block, so the signature can be fetched first. While instructions of a protected basic block are being fetched, the signature is decrypted using a key hidden in the hardware. Each fetched instruction passes through the MISR register, and the final MISR output is compared to the decrypted signature. If the calculated and the decrypted signature differ, a trap to operating system is asserted; otherwise, the instructions proceed with execution.

If the time needed to fetch a cache line from memory is greater than or equal to the decryption time, the decryption latency is hidden. The MISR calculation is completely overlapped with instruction fetch. Since the instruction addresses in the I-cache are the same as without signatures, the number of cache misses for the code with embedded signatures is the same as for the original code. Hence, the performance overhead of the SIGCED technique is due only to the additional number of processor cycles during instruction fetch, $t_{FetchOverhead}$: the number of cycles needed for address translation t_{Trans} and time needed to fetch a signature from the memory, $t_{SigFetch}$, as shown in (4). The *MemBusWidth* value is the width of the data bus between memory and the I-cache in bytes. The t_{Dbus} value is the time needed for one data bus transfer.

$$\begin{aligned} t_{FetchOverhed} &= t_{Trans} + t_{SigFetch} \\ &= t_{Trans} + \frac{SigSize}{MemBusWidth} \cdot t_{Dbus} \end{aligned} \tag{4}$$

The signature verification is done by the IBSVU, as illustrated in Fig. 6. Signature bytes are stored only in the SIGM buffer and then decrypted, while instruction bytes go to both the MISR logic and to the I-cache. An internal IBSVU signal, *sig*, controls the path of data from the data bus.

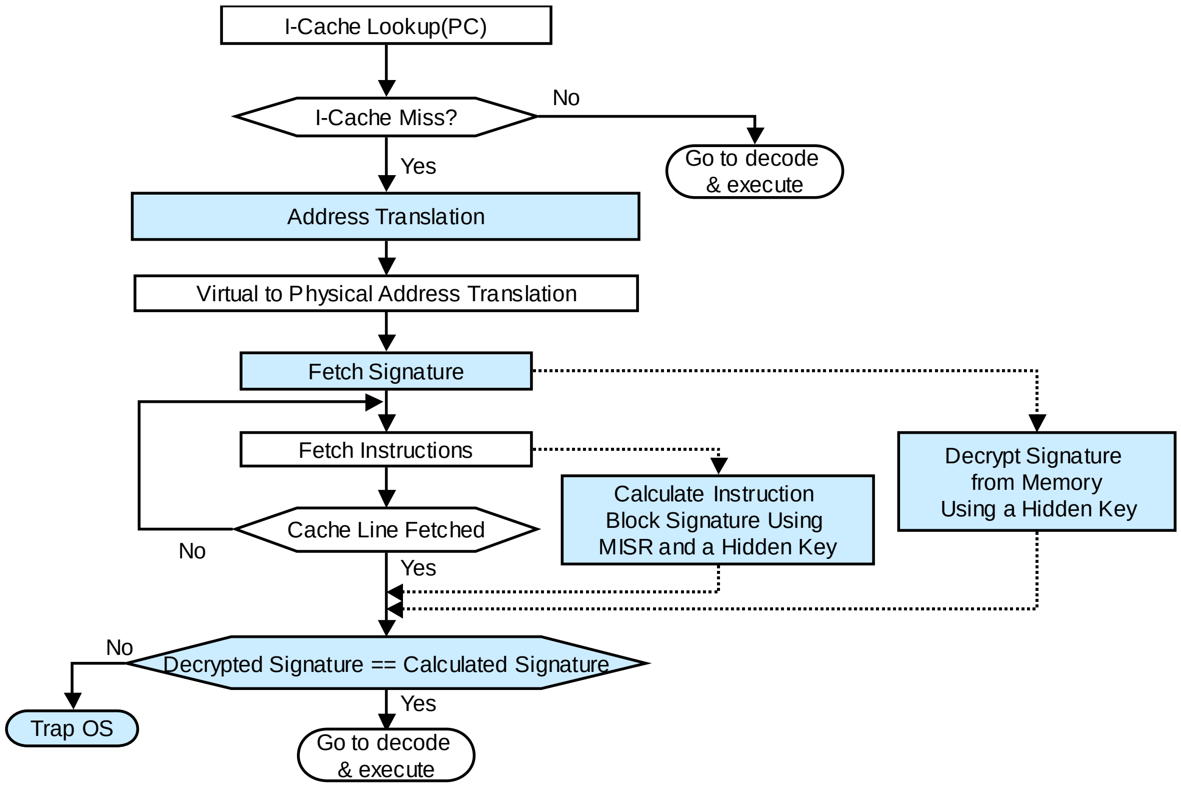
I-Cache Lookup(PC)
I-Cache Miss?
No
Go to decode & execute
Yes
Address Translation
Virtual to Physical Address Translation
Fetch Signature
Fetch Instructions
Calculate Instruction Block Signature Using MISR and a Hidden Key
Decrypt Signature from Memory Using a Hidden Key
Cache Line Fetched
No
Yes
No
Decrypted Signature == Calculated Signature
Yes
Trap OS
Go to decode & execute

Fig. 5 SIGCED: Signature verification control flow.
Dotted lines indicate parallel tasks: AES decryption and MISR calculation are done concurrently with instruction block fetch.
Shaded blocks indicate steps needed to support instruction block verification.

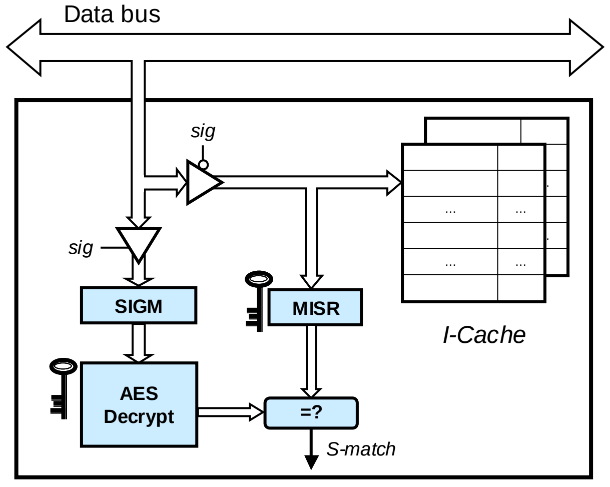

Fig. 6 SIGCED: Instruction Block Signature Verification Unit

2) *SIGCEK*

A portion of the SIGCED overhead can be avoided if signatures are not discarded after verification, but kept in a dedicated cache-like processor resource – the S-cache. Fig. 7 shows the flow of the instruction fetch process for the SIGCEK technique. With this technique, an I-cache lookup is performed together with the corresponding S-cache lookup. In the case of an I-cache miss, the instruction block signature is fetched only if it is not found in the S-cache. Otherwise, if the decrypted signature is in the S-cache, the fetch performance overhead $t_{FetchOverhead}$ in (4) is reduced to the number of cycles needed for address translation t_{Trans}. The SIGCEK technique has the potential to reduce not only the performance overhead of instruction signature verification, but also to reduce the power overhead due to signature decryption, since a cached signature is already decrypted. Fig. 8 shows a block-scheme of the IBSVU for the SIGCEK technique. A signature read from the memory and decrypted or read from the S-cache is compared to the final MISR output.

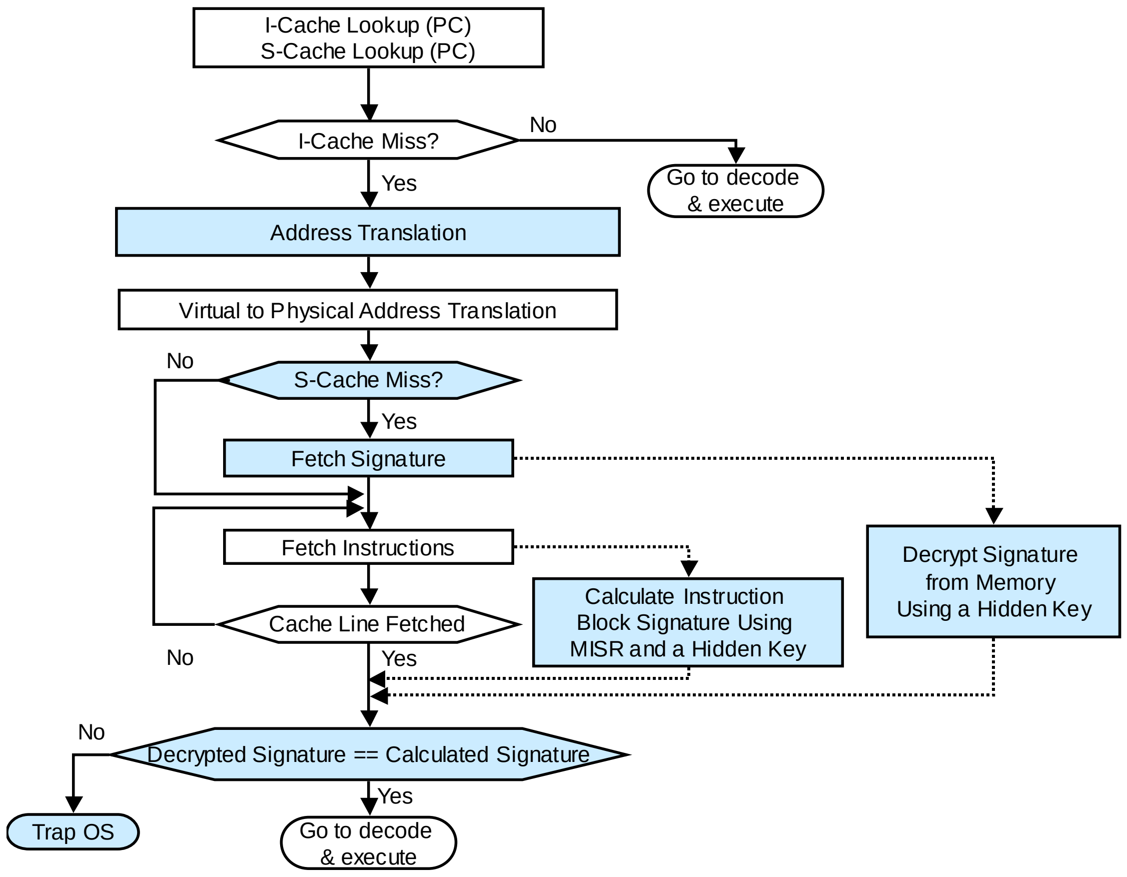

Fig. 7 SIGCEK: Signature verification control flow
Dotted lines indicate parallel tasks: AES decryption and MISR calculation are done concurrently with instruction block fetch. Shaded blocks indicate steps needed to support instruction block verification

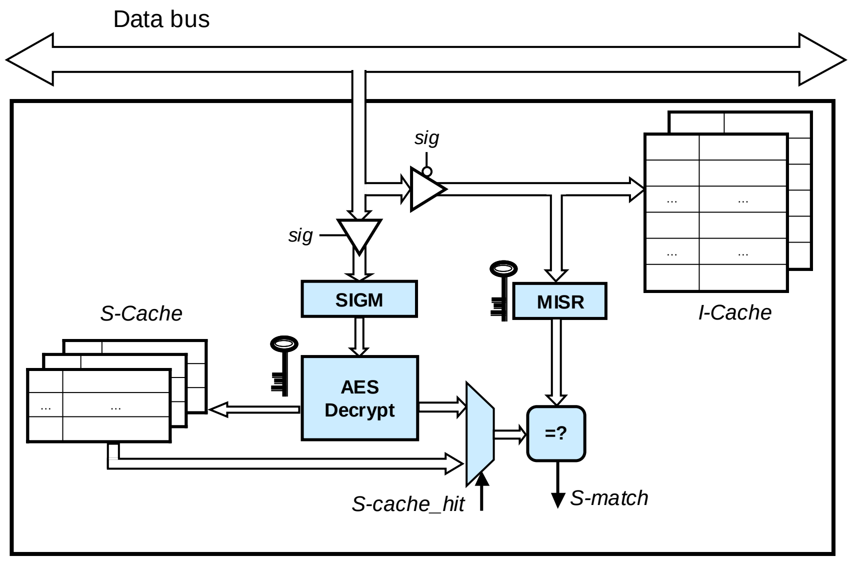

Fig. 8 SIGCEK: Instruction Block Signature Verification Unit

3) SIGCEV

In all three SIGCE techniques, the virtual addresses of instructions as seen by the processor are the same for the code with and without embedded signatures. It means that after address translation a virtual address of an instruction in the protected code is equal to the virtual address of that instruction in the original code. However, the instructions can be stored in the instruction cache with or without address translation. Two previous techniques, the SIGCED and SIGCEK, use translated virtual addresses in the cache, so both the processor and the cache see only the original virtual addresses. Another option is the SIGCEV technique: translated addresses are used only in the processor, and the instruction cache sees the non-translated virtual address space that includes the signatures and padding. Hence, in the SIGCEV, the address translation must be done before each I-cache lookup. The advantage of this technique is that the translation in most cases can be done in advance, together with the prediction of the next instruction address. The only case when the performance overhead due to the translation cannot be hidden is when a branch is mispredicted.

A simple and fast cache access mechanism requires both the cache line size and a cache line address to be a power of two. Hence, in the SIGCED and SIGCEK techniques, the size of a protected block is the power of two. Since instructions in the SIGCEV technique are stored in the I-cache using non-translated virtual addresses, the sum of sizes of a protected block and its signature is a power of two. For example, if an I-cache line size is 128 bytes and cache line addresses are aligned at 128 bytes, we can store 128 instruction bytes in one cache line in the SIGCED and SIGCEK, and 128 – *SigSize* bytes in the SIGCEV cache. Although signatures are visible to the SIGCEV I-cache, they are never stored in the cache, so the cache line size and cache size are actually smaller then the corresponding SIGCED/SIGCEK values with same cache line alignment. Another consequence of the requirement that the sum of the protected block size and signature size is a power of two is that the SIGCEV technique does not require page alignment padding, thus simplifying address translation.

III. Experimental Methodology

We analyze performance of the SIGCEK, SIGCED, and SIGCEV techniques relative to the Base configuration; the Base system does not include signature verification. As a measure of performance we use the average number of processor cycles needed for one instruction (CPI). We also consider the code size increase for different protected block sizes.

Experimental environment includes a program for emulating the secure installation and modified SimpleScalar ARM simulators [12] for each considered technique. To emulate the secure installation process, we have developed a program that embeds instruction block signatures and necessary padding in code (executable) sections of programs in the ELF format. To evaluate the proposed techniques, we modified the SimpleScalar ARM simulators to be able to execute signed ELF binaries. We also added latencies due to the signature verification and the address translation, and the S-cache for the SIGCEK.

In order to evaluate sensitivity of the proposed techniques to different system configurations, we varied the following simulation parameters:

- the I-cache size (1, 2, 4, and 8 KB);
- the I-cache line size (64 and 128 bytes);
- the width of a bus between memory and the I-cache (32 and 64 bits);
- 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 simulator parameters are shown in Table 2. We assume that the AES decryption latency with a 128-bit key is 12 cycles for slow, and 22 cycles for fast processor core, which are the speeds that can be achieved with current optimized ASIC solutions [13]. Since a signature is inserted at the beginning of the corresponding protected instruction block, signature decryption is finished before the protected block is fetched, so the decryption latency is hidden in all evaluated system configurations.

Table 2 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, FIFO replacement, first level only
I-TLB/D-TLB	32 entries, fully associative, FIFO replacement
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 mispediction latency	2 cycles for slow core, 3 cycles for fast core
TLB latency	30 cycles for slow core, 60 cycles for fast core

Table 3 Benchmark description

Benchmark	Suite	Description
blowfish_dec	MiBench	Blowfish decryption
blowfish_enc	MiBench	Blowfish encryption
cjpeg	MiBench	JPEG compression
djpeg	MiBench	JPEG decompression
ecdhb	Basicrypt	Diffie-Hellman key exchange
ecdsignb	Basicrypt	Digital signature generation
ecdsverb	Basicrypt	Digital signature verification
ecelgdecb	Basicrypt	El-Gamal decryption
ecelgencb	Basicrypt	El-Gamal encryption
ispell	MiBench	Spell checker
mpeg2_enc	MediaBench	MPEG2 compression
qsort	MiBench	Quicksort
rijndael_dec	MiBench	Rijndael decryption
rijndael_enc	MiBench	Rijndael encryption
stringsearch	MiBench	String search

We used benchmarks from several benchmark suites for embedded systems: MiBench [14], MediaBench [15], Basicrypt [16] (Table 3). All benchmarks but *mpeg2_enc* use the largest possible provided input. *Mpeg2_enc* uses the provided test input. Table 4 shows the total size of the original binary and the total size of the executable code sections in bytes, and the number of executed instructions. In a binary file with embedded signatures, only the size of executable code sections will change, depending on the technique used and the size of the protected blocks and signatures.

Since the 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. Table 5 shows the number of I-cache misses per 1000 instructions, for cache lines of 64 and 128 B, and cache sizes of 1, 2, 4, and 8 KB. Since all benchmarks have very low I-cache miss rate in a 8 KB cache, this was the largest cache size that we simulated. The size of the MISR and generated signatures is 128 bits (16 bytes), which is a minimum size for AES encryption.

Table 4. Benchmark code size and executed instructions

Benchmark	Code size [B]	Text segment size [B]	Executed instructions [million]
blowfish_dec	1,032,731	190,900	544.0
blowfish_enc	1,032,731	190,900	544.0
cjpeg	1,261,485	298,916	104.6
djpeg	1,274,670	311,108	23.4
ecdhb	1,102,298	258,188	122.5
ecdsignb	1,254,373	310,068	131.3
ecdsverb	1,254,519	310,212	171.9
ecelgdecb	1,102,207	258,092	92.4
ecelgencb	1,102,271	258,156	180.2
ispell	1,238,144	240,972	817.7
mpeg2_enc	1,318,326	317,504	127.5
qsort	1,180,697	252,284	737.9
rijndael_dec	1,045,273	199,364	307.9
rijndael_enc	1,045,273	199,364	320.0
stringsearch	1,025,446	188,484	3.7

Table 5. I-cache misses per 1000 instructions for the Base case

Benchmark	Cache line 64 B				Cache line 128 B			
	1 K	2 K	4 K	8 K	1 K	2 K	4 K	8 K
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

IV. RESULTS

We first evaluated the code size increase due to embedded signatures. For 128 B cache lines, the signatures increase the size of executable code sections by 12.5% for the SIGCED and SIGCEK techniques. With the SIGCEV the executable code is increased by 14.28%. However, an ELF binary typically includes other sections, such as headers, initialized data, symbol table, and debugging information, so the impact of signatures on the total binary size will be smaller. In our benchmarks compiled with the ARM gcc cross-compiler and the *–static* option, the code sections do not make more than 25% of the total binary, so the total program size increases from 2.6 to 3.5% only. With 64 B cache line, the total program size for considered benchmarks increases 4.7-6.3% for the SIGCED and 6.1-8.3% for the SIGCEV technique.

The results of the performance analysis are presented as follows. We first analyze the performance overhead of the SIGCED, SIGCEK, and SIGCEV techniques assuming the initial configuration with a slow processor core and a 32-bit memory data bus. Next, we examine performance of the SIGCEK technique as a function of the size of the S-cache. Finally, we explore how performance of the SIGCED and SIGCEV techniques depends on changes of the processor core speed, memory bus width, and the size of protected blocks.

A. Performance overhead with slow core, 32-bit bus, and 128 B I-cache line

Fig. 9 and Fig. 10 show the SIGCED, SIGCEK, and SIGCEK CPI normalized to the Base case, for I-cache sizes of 1 K and 4 K. The CPI for the Base case is given in Table 6.

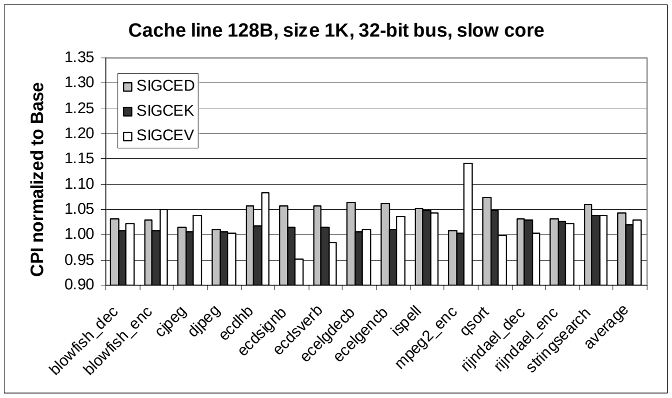

Fig. 9 CPI normalized to the Base for slow processor core, 32-bit memory bus, cache size 1 K, cache line 128 B

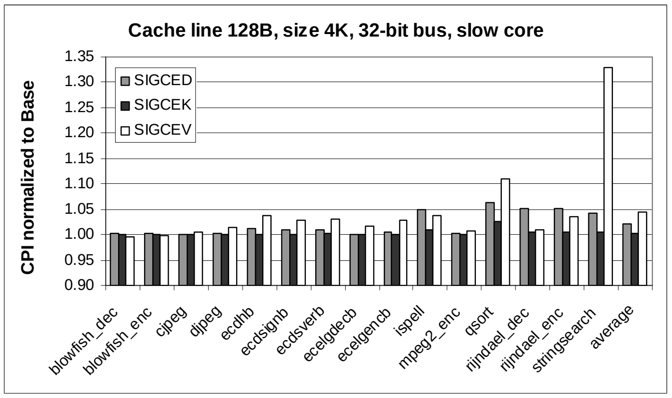

Fig. 10 CPI normalized to the Base for slow processor core, 32-bit memory bus, cache size 4 K, cache line 128 B

The results indicate a very low performance overhead of the SIGCED technique. Even with the small 1 K I-cache, this technique increases CPI in the range 0.8-7.4%, with 8 out of 15 benchmarks having more than 5% increase. With the 4 K I-cache, CPI increases for more than 5% for only 3 benchmarks, since the influence of signature verification overhead is reduced with I-cache miss reduction.

The absolute CPI increase for the SIGCED technique depends on the number of I-cache misses given in Table 5: more cache misses means more signature verifications, that is, increased performance overhead. However, the CPI normalized to the Base case does not follow the number of I-cache misses, since for an application with a relatively large number of I-cache misses a relative CPI increase may be smaller than for an application with fewer cache misses. For example, with the 1 K I-cache *rijndael_enc* has a 3% CPI increase and *ecdhb* has a 5.8% increase, whereas *rijndael_enc* has 42.58 I-cache misses per 1000 instructions, and *ecdhb* only 14.57. This can be easily explained by the fact that the Base CPI for this system configuration is 15.51 for *rijndael_enc* and 3.28 for *ecdhb*, and the absolute CPI increase with the SIGCED technique is 0.19 for *ecdhb* and 0.47 for *rijndael_enc*.

Table 6 CPI for the Base configuration, cache line 128 B, memory bus 32 bits, slow processor core

Benchmark	**1 K**	**2 K**	**4 K**	**8 K**
blowfish_dec	5.49	4.55	3.03	2.30
blowfish_enc	5.42	4.55	3.03	2.30
cjpeg	5.21	3.64	2.99	1.81
djpeg	8.39	5.17	3.41	1.86
ecdhb	3.28	2.28	1.77	1.62
ecdsignb	3.36	2.21	1.81	1.70
ecdsverb	3.37	2.26	1.83	1.71
ecelgdecb	4.20	1.99	1.78	1.78
ecelgencb	3.80	2.14	1.77	1.70
ispell	9.57	7.93	5.48	2.65
mpeg2_enc	3.39	2.42	1.75	1.52
qsort	5.59	3.97	3.16	2.31
rijndael_dec	15.65	12.88	9.06	4.47
rijndael_enc	15.51	12.51	8.89	4.41
stringsearch	7.95	5.57	3.18	2.00

As explained in Section II, the SIGCED overhead can be reduced if signatures are kept in the S-cache, i.e., with the SIGCEK technique. In order to be able to keep some signatures when the corresponding protected blocks are evicted from the I-cache, the S-cache must have greater associativity and/or more entries than the I-cache. The S-caches in Fig. 9 and Fig. 10 have twice as many entries as the corresponding I-caches, fully associative organization, and the LRU replacement policy. The total size of the S-cache is ¼ of the I-cache size. The SIGCEK CPI increase is in the range 0.3-4.8% with the 1 K I-cache and 0.006-2.5% with the 4 K I-cache. The SIGCEK reduces the performance overhead of the SIGCED for 8-91% and 50-92% with the 1 K and 4 K I-cache, respectively.

The normalized CPI for the SIGCEV technique is in the range 0.95-1.14 with 1 K I-cache, and 0.99-1.33 with the 4 K I-cache. The SIGCEV protected block size in these experiments is 112 B, so the actual I-cache size is 0.875 of the Base I-cache size. Since the SIGCEV I-cache is smaller, for most benchmarks it has more cache misses than the Base I-cache (Table 5, Table 7). The large SIGCEV performance overhead of 14% for *mpeg2_enc* with 1 K I-cache and 33% for *stringsearch* with the 4 K I-cache is due to the significant relative increase in the number of cache misses.

However, the SIGCEV may have even a lower CPI than the Base case. The SIGCEV I-cache has a different mapping function, so the number of I-cache misses may be lower, especially in small caches with more capacity misses. If such a benchmark also has a relatively low branch misprediction rate, the SIGCEV might marginally outperform the Base case. This is the case with the *ecdsignb* and *ecdsverb* benchmarks with the 1 K I-cache, and *blowfish* with the 4 K I-cache. It should be noted that the difference in the number of I-cache misses between the Base case and the SIGCED may be reduced if both the original code and the code with signatures are transformed to optimally use available cache resources, as described in [17].

It is interesting to note that the SIGCEV technique outperforms the SIGCED for 11 out of 15 benchmarks with the 1 K I-cache. This is due to the relatively large number of cache misses in such a small cache, and to the different instruction block address translation in the SIGCED and SIGCEV techniques. With the SIGCEV, the address translation latency is hidden except when a branch is mispredicted, and it is never hidden with the SIGCED. Hence, the overhead of the increased number of I-cache misses in the SIGCEV may be less than the overhead of the address translation in the SIGCED. For example, for *ecelgdecb* and 1 K I-cache the SIGCED has 4.48 CPI, and the SIGCEV has 4.23 CPI. The number of I-cache misses is 22.4 per 1000 instructions with the SIGCED, and only slightly larger with the SIGCEV, 23.7 per 1000 instructions.

Table 7. I-cache misses per 1000 instructions for SIGCEV

Benchmark	Cache line 64 B				Cache line 128 B			
	1 K	2 K	4 K	8 K	1 K	2 K	4 K	8 K
blowfish_dec	29.2	16.7	0.1	0.0	14.9	9.8	0.8	0.0
blowfish_enc	28.5	14.4	3.1	0.0	15.0	6.9	0.8	0.0
cjpeg	10.8	3.3	0.4	0.1	8.8	2.1	0.3	0.1
djpeg	21.5	6.9	2.6	0.3	7.0	3.5	1.5	0.2
ecdhb	30.7	13.7	4.9	0.5	17.2	8.7	2.2	0.2
ecdsignb	24.2	10.8	3.8	0.4	13.5	6.7	1.7	0.2
ecdsverb	25.9	11.5	4.2	0.7	14.5	7.2	1.9	0.3
ecelgdecb	40.5	9.1	0.2	0.0	23.7	6.9	0.1	0.0
ecelgencb	35.8	11.5	2.6	0.3	20.5	7.8	1.2	0.1
ispell	76.1	65.8	32.8	6.9	48.1	42.9	23.3	5.9
mpeg2_enc	8.7	1.3	0.6	0.2	6.9	0.8	0.3	0.1
qsort	52.4	38.9	29.0	13.5	31.7	25.1	18.0	9.7
rijndael_dec	89.7	85.8	85.8	33.8	44.2	41.6	40.3	10.0
rijndael_enc	88.9	87.0	86.4	45.6	47.0	43.8	41.3	22.9
stringsearch	69.0	43.6	19.5	0.3	44.4	32.9	20.8	5.5

B. Sensitivity of SIGCEK to the S-cache size

The very low performance overhead of the SIGCEK technique with the S-cache parameters as described in the previous section prompted us to experiment with a smaller S-cache. We call the S-cache used in the previous experiments (Fig. 9, Fig. 10) the medium-size S-cache. A small-size S-cache in our further experiments is half of the size of the medium S-cache. That is, the small S-cache has the same number of entries as the I-cache, fully associative organization, and the LRU replacement policy. The size of the small S-cache is 1/8 of the corresponding I-cache size.

Fig. 11 shows the CPI for the SIGCEK with the small S-cache normalized to the SIGCEK CPI with the medium S-cache, for two system configurations (1 K and 4 K I-cache). Although investing more hardware resources into the S-cache always reduces performance overhead, we can observe that in some cases the medium-size S-cache only marginally outperforms the small S-cache, e.g., for *cjpeg* with the 4 K I-cache and for *rijndael_dec* with the 1 K I-cache. This can happen due to one of the two following reasons. If a benchmark has a very low I-cache miss rate for a certain cache size, e.g. *cjpeg* with 4 K I-cache (Table 5), employing a medium-size S-cache will not significantly influence the CPI, since most cache misses are cold misses. If a benchmark has a relatively large miss rate for both the considered cache size and the next larger size, e.g., *rijndael_dec* with the 1 K I-cache, the medium-size S-cache impact will also be low, since it will not be able to retain the signatures of evicted cache blocks. In the future work we plan to explore possible ways to increase the S-cache hit rate. One solution would be to use a different replacement policy, for example to replace a signature whose corresponding I-cache block was the least recently evicted.

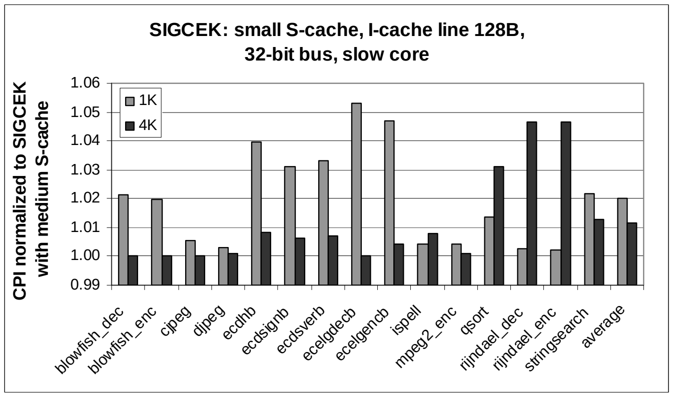

Fig. 11 SIGCEK: small vs. medium-size S-cache

C. Sensitivity of SIGCED and SIGCEV to core speed, memory bus width, and protected block size

The number of processor cycles needed for the signature fetch will decrease with the wider data memory bus, and increase with the faster processor core, so we may expect similar behavior from total signature verification overhead. Another interesting parameter is the cache line size. Although the size of a program with shorter protected blocks will increase more due to embedded signatures, such program may have a lower CPI if the number of I-cache misses is reduced with shorter cache lines. We performed experiments with all possible variations of these parameters, that is, with slow and fast core, 32- and 64-bit memory bus, 64 and 128 B I-cache block, and various I-cache sizes.

We may group the benchmarks in two groups according to the number of cache misses with all considered cache sizes. The influence of the bus width, the core speed, and the cache line size will be discussed for one benchmark from each group: *ecdhb* with a relatively low number of cache misses, and *rijndael_enc* which is one of the two benchmarks with the largest number of cache misses per 1000 instructions for each cache size and line size (Table 5).

Fig. 12 and Fig. 13 show the SIGCED CPI normalized to the Base case for *ecdhb* and *rijndael_enc*. As expected, the SIGCED technique has the largest impact on performance with the 64 B cache line size, the 32-bit bus, and a fast processor core. However, even with this system configuration the SIGCED performance overhead is never more than 13% for both benchmarks. If the number of I-cache misses is very low, as it is for *ecdhb* in the 8 K I-cache, the SIGCED overhead does not depend on system parameters, since it is always close to zero.

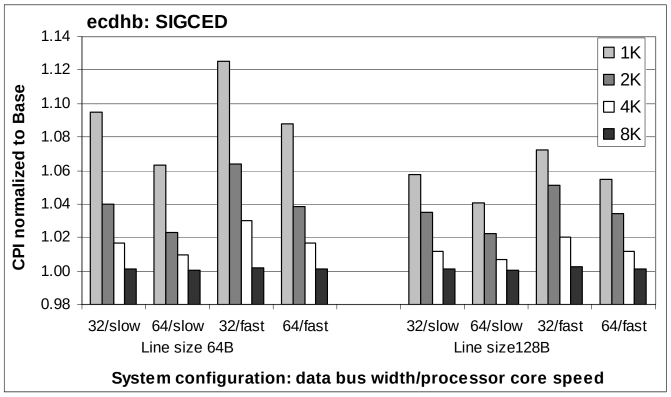

Fig. 12 SIGCED: CPI normalized to the Base, ecdhb

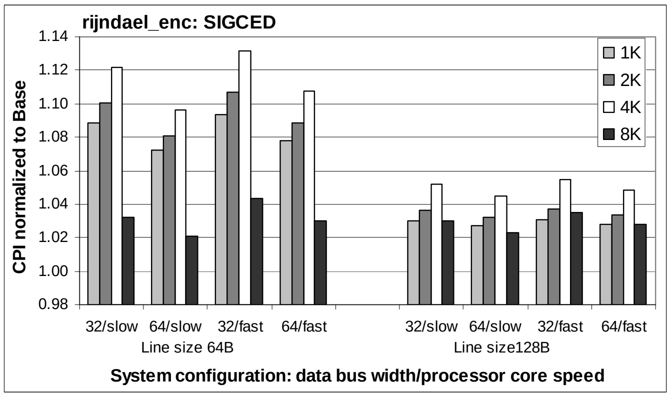

Fig. 13 SIGCED: CPI normalized to the Base, rijndael_enc

It is interesting to note that the normalized SIGCED CPI decreases with larger caches for *ecdhb* and not for *rijndael_enc*. The *rijndael_enc* benchmark has a very large number of I-cache capacity misses in 1, 2, and 4 K caches, such that the number of I-cache misses is only slightly reduced with the cache size increase before the 8 K size (Table 5). Hence, the absolute overhead of the SIGCED technique does not considerably decrease with the cache size increase. However, even a relatively small reduction in the number of cache misses significantly improves the Base CPI, so the normalized CPI for the SIGCED actually grows, up to the 4 K cache size.

Fig. 14 and Fig. 15 show the SIGCEV CPI normalized to the Base case for *ecdhb* and *rijndael_enc*. For both benchmarks the SIGCEV has more I-cache misses than the Base case (Table 5, Table 7), so it always has lower performance. For both benchmarks the SIGCEV is more sensitive to configuration change than the SIGCED, since a narrower bus and a faster core increase both the cache miss latency and the latency due to signature fetch. Similarly to the SIGCED, the SIGCEV with longer cache line has smaller performance overhead.

The *rijndael_enc* benchmark has a very large SIGCEV performance overhead with the 8 K I-cache. This happens because the number of I-cache misses has doubled compared to the Base case. Moreover, there is a sharp drop in the Base CPI for 8 K I-cache (Table 6), so the relative CPI increase is even more noticeable.

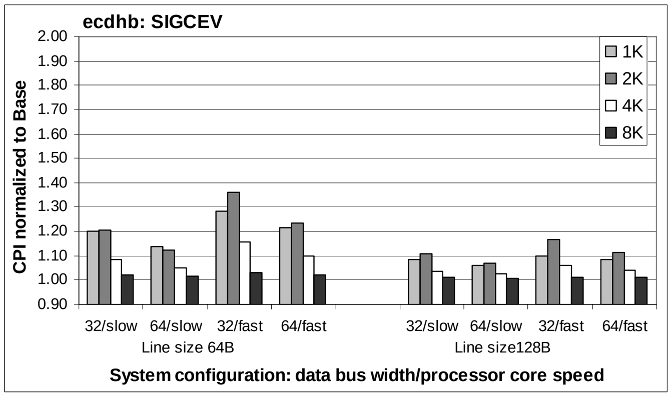

Fig. 14 SIGCEV: CPI normalized to the Base, ecdhb

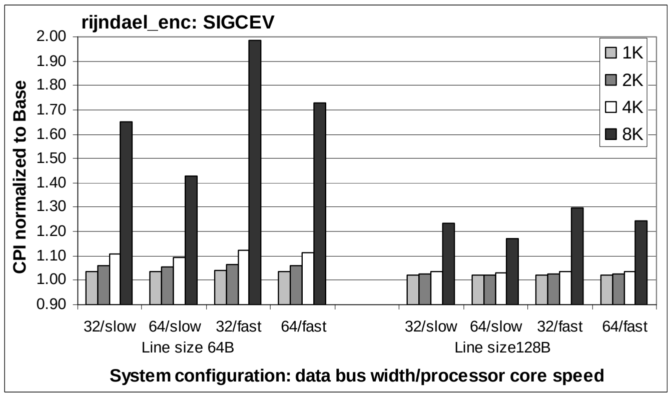

Fig. 15 SIGCEV: CPI normalized to the Base, rijndael_enc

We may conclude that if an embedded system has a low hardware budget, the SIGCEV technique has the best price-performance tradeoff, since in small caches it outperforms the SIGCED for most benchmarks and employs less hardware resources. However, the SIGCED is better for systems with larger caches. With 25% larger hardware budget invested in the S-cache, the SIGCEK technique has a very low performance overhead across all considered system configurations.

V. Related Work

Techniques for countering code injection attacks can be classified in two categories: those that are software-based and those that require some hardware support. The software techniques can be further classified into static techniques and dynamic techniques.

Static code analysis can find a significant number of security flaws and suggest where changes in the code should be made. However, it is impossible to discover all vulnerabilities in any given program, since the problem of static analysis is generally undecidable [18]. 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 [19]. The need for precise automated analysis can be alleviated if programmers adds specially formulated comments about constraints [20], [21], but adding annotations can be as error prone as programming itself and puts additional burden on programmers.

Dynamic software techniques encompass several groups of techniques that detect and/or prevent attacks in run-time. One group of techniques augments the code with various run-time checks [10], [22], [23], [24], [25]. Another group comprises of “safe dialects” of language C, which restrict the use of unsafe constructs, perform static analysis and/or runtime checks, and use garbage collection or region-base memory management [26], [27], [28]. Another approach is obfuscation: segment addresses, jump addresses, or the complete code can be scrambled, making it difficult for an attacker to succeed [29], [30], [31], [32]. Several researchers suggest intrusion detection by monitoring the program behavior, such as monitoring the sequences of system calls of a program [33], [34], or the values of monitoring performance registers [35]. 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 instructions executed, they may incur significant performance and power overhead.

Some of the performance overhead may be reduced with hardware support. Xu et al. propose an architectural support against the stack smashing attack: a return address is saved on both the Secure Return Address Stack and on the “regular” stack [36]. An attack is detected if the two addresses do not match. Similar efforts expand this idea [37], [38]. [39]. The secure stack does not have to be implemented in hardware: with the Dynamic Instruction Stream Editing (DISE), the ‘shadow” stack is kept in a protected area on the heap [40]. DISE is a one-to-many instruction macro expander with programmable rewriting rules: to protect return addresses from the attack, call and return instructions are dynamically rewritten in the runtime to write/verify data from the shadow stack. Another approach is to achieve redundancy of return addresses not by duplicating stack, but by replicating cache lines with return address [41]. When a return is executed, the value of replicated data is compared to the return address. The main drawback of these techniques is that they provide protection from only one type of attack. Techniques such as specific randomized instruction sets for each process may prevent code injection in general [42], but at the price of a significant increase in execution time.

A successful buffer overflow attack can overwrite not only return addresses on the stack, but any function pointer. Tuck et al. [43] propose to protect code pointers from both read and write buffer overflow attacks by encrypting them. Code pointers are encrypted and decrypted using special instructions, encrypt-stores and decrypt-loads. The authors propose three levels of encryption: XOR with a secret key, XOR with a value from random permutation table, or a Feistel network. Decrypted values are cached in the L1 cache memory. Another technique, HSAP, also encrypts function pointers, but only using a simple XOR with a secret key [44]. The HSAP also protects from stack smashing by implementing a hardware boundary check for stack variables.

A framework that encompasses secure installation and secure program execution was first proposed by Kirovski et al. (Secure Program Execution Framework, SPEF [45]). However, our work expands their initial idea and offers a more efficient implementation. Kirovski et. al do not store the calculated instruction block signature as we do, but transform instruction blocks according to the encrypted hash values of transformation invariants. During installation, a transformation-invariant (TI) hash value is calculated for each instruction block and is encrypted using AES and a secret processor key; the encrypted hash value determines the transformation of the instruction block. The chosen transformation belongs to a set of transformations that do not change the correct program behavior, such as instruction scheduling, basic block reordering, branch-type selection, and register permutation. During execution, the verifier component calculates the TI hash for every instruction block that is fetched after an I-cache miss. It then encrypts the hashed value, and verifies whether the obtained transformation is equal to the actual code. The performance overhead can be reduced if TI hash values are kept in the TI-cache. The advantage of this approach is a minimal increase in the code size. However, our techniques are less complex than SPEF, so they have less performance overhead: for MediaBench benchmarks, the authors report overhead 12.7-24.7%, and 7.5-17% with a TI-cache. Moreover, on average there is less possible transformations of a given instruction block than possible signatures generated with our approach, so using 16-byte signatures appears to be more cryptographically secure. Finally, some of the SPEF code transformations require compiler support, whereas our signature techniques do not.

Another interesting approach is to tag all data coming from “the outside world” (e.g., I/O channels) as spurious and to prevent execution of any control transfer instruction if the target address depends on spurious data [46]. This approach may generate some false positives, since the target address may be input-dependant, for example in switch constructs. Generally, input data can propagate to a target address through a series of calculations, so this technique requires a relatively complex data dependency analysis. A similar approach, Minos, [47] augments every memory word with an integrity bit. The integrity bit is set by kernel and determines the trust the kernel has in that data. The low trust data cannot be used for control transfers.

The code integrity in run-time can be successfully protected if all instruction blocks are signed with a cryptographically secure signature. We did a preliminary research on protection of basic blocks and cache blocks using signatures [8], [9]. Kirovski et al. also propose to sign all cache blocks and to verify signatures in run-time [48]. An instruction block signature is obtained by encrypting the instruction block using a 128-bit Rijndael cipher, and then XOR-ing the 16-byte sub-blocks. The overhead of Rijndael decryption implemented in hardware can be hidden if the instructions in an instruction block can be reordered in such a way that critical instructions such as stores are executed after decryption delay time.

Signatures of instruction blocks of various granularity are frequently used in fault-tolerant computing [49], and Joseph and Avizienis proposed the idea of a virus protection technique using an extended Program Flow Monitor [50]. However, the paper does not include any implementation details or evaluation.

VI. Conclusion

In this paper we propose and analyze three hardware-supported techniques for runtime instruction block verification: SIGCED, SIGCEK, and SIGCEV. These techniques provide complete software integrity with minimal to modest hardware investments and no compiler support. If chosen according to the available hardware budget, the three proposed techniques do not impose significant burden on the overall performance. The SIGCEV should be implemented with small caches, the SIGCED with a medium-size hardware budget, and the SIGCEK if the budget allows the S-cache.

Another contribution of this paper is a taxonomy of all possible techniques for runtime instruction block verification. Techniques are classified according to the type of protected instruction blocks (variable vs. fixed size), placement of instruction block signatures (embedded vs. table), signature handling (dispose after verification or keep in the S-cache), and signature visibility (visible vs. hidden from the I-cache).

Future work will evaluate the power overhead of the proposed techniques. We will also evaluate other possible implementations, such as protecting more than one cache block with the same signature, and using other replacement policies for the S-cache. Another interesting question is whether the proposed basic mechanism can be extended to cover other classes of attacks, such as return-into-libc. Although the main goal of the proposed mechanism is to prevent code injection attacks, it can be applied to other purposes, such as fault-tolerant execution, virus protection, and protection from software tampering.

References

[1] A. One, "Smashing the Stack for Fun and Profit," *Phrack Magazine*, vol. 7, November 1996.

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

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

[4] US-CERT, "Cyber Security Bulletin Sb04-175," <http://www.us-cert.gov/cas/body/bulletins/SB04-175_H.html> (Available November 2004).

[5] L. Garber, "New Chips Stop Buffer Overflow Attacks," *IEEE Computer*, vol. 37, October 2004, pp. 28.

[6] T. Alves and D. Felton, "Trustzone: Integrated Hardware and Software Security," *I.Q. Publication*, vol. 3, November 2004.

[7] NIST, "Fips Pub 197: Advanced Encryption Standard (AES)," 2001.

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

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

[10] M. Prasad and T.-c. Chiueh, "A Binary Rewriting Defense against Stack-Based Buffer Overflow Attacks," in *Usenix Annual Technical Conference*, San Antonio, TX, USA, 2003, pp. 211-224.

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

[12] D. Burger and T. Austin, "The SimpleScalar Tool Set Version 2.0," University of Wisconsin, Technical Report CS-TR-97-1342, 1997.

[13] "Enhanced AES (Rijndael) Ip Core," <http://www.asics.ws> (Available December 2004).

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

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

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

[17] S. Bartolini and C.A. Prete, "A Cache-Aware Program Transformation Technique Suitable for Embedded Systems," *Information and Software Technology*, vol. 44, 2002, pp. 783-795.

[18] W. Landi, "Undecidability of Static Analysis," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, December 1992, pp. 323-337.

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

[20] D. Larochelle and D. Evans, "Statically Detecting Likely Buffer Overflow Vulnerabilities," in *10th USENIX Security Symposium*, Washington, DC, USA, 2001, pp. 177-189.

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

[22] J. L. Steffen, "Adding Run-Time Checking to the Portable C Compiler," *Software—Practice & Experience*, vol. 22, April 1992, pp. 305-316.

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

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

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

[26] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A Safe Dialect of C," in *USENIX Annual Technical Conference*, Monterey, CA, USA, 2002, pp. 275-288.

[27] G. C. Necula, S. McPeak, and W. Weimer, "CCured: Type-Safe Retrofitting of Legacy Code," in *29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Portland, OR, USA, 2002, pp. 128-139.

[28] D. Dhurjati, S. Kowshik, V. Adve, and C. Lattner, "Memory Safety without Runtime Checks or Garbage Collection," in *2003 ACM SIGPLAN Conference on Language, Compiler, and Tool Support for Embedded Systems*, San Diego, CA, USA, 2003, pp. 69-80.

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

[30] P. Busser, "Memory Protection with PaX and the Stack Smashing Protector: Breaking out Peace," *Linux Magazine* March 2004, pp. 36-39.

[31] E. G. Barrantes, D. H. Ackley, T. S. Palmer, D. Stefanovic, and D. D. Zovi, "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," in *10th ACM Conference on Computer and Communication Security*, Washington, DC, USA, 2003, pp. 281-289.

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

[33] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," in *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 1999, pp. 133-145.

[34] R. Sekar, T. Bowen, and M. Segal, "On Preventing Intrusions by Process Behavior Monitoring," in *8th USENIX Security Symposium*, Washington, DC, USA, 1999, pp. 29-40.

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

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

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

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

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

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

[41] K. Inoue, "Energy-Security Tradeoff in a Secure Cache Architecture against Buffer Overflow Attacks," in *Workshop on Architectural Support for Security and Anti-Virus (WASSA)*, Boston, MA, USA, 2004, pp. 77-85.

[42] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering Code-Injection Attacks with Instruction-Set Randomization," in *10th ACM Conference on Computer and Communication Security*, Washington, DC, USA, 2003, pp. 272-280.

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

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

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

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

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

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

[49] A. Mahmood and E. J. McCluskey, "Concurrent Error Detection Using Watchdog Processors-a Survey," *IEEE Transactions on Computers*, vol. 37, Feb. 1988, pp. 160-174.

[50] M. K. Joseph and A. Avizienis, "A Fault Tolerance Approach to Computer Viruses," in *IEEE Symposium on Security and Privacy*, Oakland, CA, USA, 1988, pp. 52-58.