Using Instruction Block Signatures to Counter Code Injection Attacks

Milena Milenković, Aleksandar Milenković, Emil Jovanov
Electrical and Computer Engineering Department, the University of Alabama in Huntsville
Email: {milenkm | milenka | jovanov}@ece.uah.edu

ABSTRACT

With more computing platforms connected to the Internet each day, computer system security has become a critical issue. One of the major security problems is execution of malicious injected code. In this paper we propose new processor extensions that allow execution of trusted instructions only. The proposed extensions verify instruction block signatures in run-time. Signatures are generated during a trusted installation process, using a multiple input signature register (MISR), and stored in an encrypted form. The coefficients of the MISR and the key used for signature encryption are based on a hidden processor key. Signature verification is done in the background, concurrently with program execution, thus reducing negative impact on performance. The preliminary results indicate that the proposed processor extensions will prevent execution of any unauthorized code at a relatively small increase in system complexity and execution time.

1. INTRODUCTION

Most of today's computers are connected to the Internet or at least to a local network, exposing system vulnerabilities to potential attackers. Consequently, computer security is becoming a critical issue, and current trends in hardware and software will bring it even more into focus. Following Moore's law, in the next five years we can expect high-end processors with one billion transistors, and proliferation of Internet-enabled, low-end embedded systems, ranging from home appliances, cars, and sensor networks to personal health monitoring devices. Increased complexity of high-end systems and the large-scale deployment and diversity of low-end systems make it difficult to uncover system vulnerabilities. In addition, exhaustive testing is virtually impossible as software grows in size and complexity and time-to-market decreases.

One of the major security problems is execution of unauthorized and potentially malicious code. During execution of vulnerable programs an attacker is able to inject the code into a memory structure, for example a buffer, and then to change the code pointer, such as the return value on the stack [1]. One such example is the so-called "stack smashing": an attacker exploits a possibility for a buffer overflow in the program by sending more data than the buffer can hold, overwriting the valid return address on the stack with the malicious code address and writing the malicious code also on the stack. When this code is executed, it will have the same privileges as the attacked program. Various other examples of attacks exist, such as heap overflow and format string attacks [2].

The ever-increasing available area on a chip so far has predominantly been used for faster execution. With more complex software having potentially a larger number of defects, increased number of attacks, and proliferation of networked computing platforms, we believe that dedicated processor resources should be used to provide more secure execution. Hardware-supported techniques have potential to provide secure program execution with lower overhead in performance, cost, and power consumption than techniques relying solely on software.

In this paper we propose processor extensions that allow execution of trusted instructions only, by verifying instruction block signatures in run-time. An instruction block signature is determined during secure program installation, using a multiple input signature register (MISR) with linear feedback coefficients dependent on a secret processor key. Only instruction blocks that caused cache misses need to be verified, since an instruction cache is a read-only structure.

We consider three implementations of this mechanism: SIGT, SIGE, and SIGC. In the SIGT and SIGE, an atomic code unit protected by its signature is a basic block. In the SIGT signatures are stored in a signature table in a separate code segment [3], and in the SIGE signatures are

embedded in the code, so that each basic block contains its signature. The SIGT implementation requires more hardware resources, such as the cache-like structure for storage of the most recently needed signatures (IBST), while the SIGE requires an additional opcode or reserved instruction bit for signatures and increases the number of cache misses. Both SIGE and SIGT require compiler support to determine the list of basic blocks.

The SIGC is a more efficient variation of the SIGE: instruction block signatures are also embedded in the code, but the size of an instruction block corresponds to a cache block size, not to a basic block. All signatures are stored at predetermined addresses, so there is no need for additional opcode or for compiler support. Although both SIGE and SIGC techniques store instruction block signatures in main memory and verify them at each instruction cache miss, in the SIGC signatures are not stored in the cache, since they are not needed after successful verification.

The potential of proposed techniques is evaluated using SPEC CPU2000 benchmarks. Most results indicate a minor increase in the execution time, at a relatively modest hardware cost. For the SIGT, very few applications have more than 1000 IBST misses per one million instructions, for as low as 64 IBST entries. A convenient hash function can minimize the number of memory accesses on an IBST miss. For a reasonable instruction cache size of 32K, the number of misses is relatively low, so the increased code size in the SIGE does not significantly increase the absolute number of cache misses for most considered applications. The increase in the number of cache misses can be completely avoided with the SIGC, with very small increase in IPC, up to 0.075.

We believe that the overhead of the architectural extensions is a small price to pay for added security. Instead of the vulnerability-specific solutions, the proposed implementations offer protection from a whole class of vulnerabilities that allow execution of a malicious code. The proposed extensions are cost-effective, do not require significant processor changes and changes in legacy source code. In addition, encrypted basic block signatures protect the code from software tampering, and enable fault detection in error-prone environments such as Space.

This paper is organized as follows. Section 2 describes the related work, and Section 3 describes the proposed techniques. Section 4 shows the preliminary results, and the last section concludes the paper.

2. RELATED WORK

One obvious but unattainable solution to the problem of injected code execution would be to write code that is not vulnerable to such attacks. Instead, we rely on various defense techniques that 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, which detect security defects in the code in compile time, and dynamic techniques, which augment the program to detect the execution of unauthorized code in run time.

Static code analysis can find a significant number of security flaws and suggest where changes in the code should be made. However, the problem of static analysis is generally undecidable [4]. 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. Wagner et al. proposed a tool for automated detection of code that might cause the buffer overflow [5]. The problem of buffer overflow is formulated as an integer constraint problem: a string buffer is modeled as a pair of integers, one for the current buffer length and another for the allocated size, so the tool needs to verify whether the maximum length is not greater than the allocated size. The authors admit they sacrificed precision in order to have a scalable tool. The need for precise automated analysis can be alleviated if the programmer adds specially formulated comments about constraints [6]. In a recent study, Dor et al. propose a tool for detection of all string manipulation errors, C String Static Verifier [7]. This tool is able to find all such errors, providing that the potentially vulnerable functions are annotated with so-called contracts, including pre-conditions, post-conditions, and potential side effects. The authors also propose algorithms for automated strengthening of post- and pre-conditions, reducing the burden placed on the programmer, but at the cost of increased imprecision.

Dynamic software techniques augment the code with run-time attack detection. Most of these techniques concentrate on one type of known attack, especially stack smashing. For example, the StackGuard compiler places a dummy value, the so-called canary, between the return address and the rest of the stack [8]. A buffer overflow attack that overwrites the return address must also overwrite the canary. Hence, an attack is detected if the value of the canary has changed. Another approach is to check the range of referenced buffers in the function wrapper [9]. Run-time detection can be applied only to the critical library functions, such as string manipulation functions or malloc() [10], or to the whole program, using modified C compilers [9, 11, 12] or "safe dialects" of the C language [13]. Binary code can be directly modified [11], but these techniques may have some false negatives or positives. Protection can also be implemented at the level of the operating system [14], and there are several open source Linux distributions with security features, such as Hardened Gentoo, Kaladix Linux, Openwall, and RedHat [15]. One interesting approach is to obfuscate the addresses: the virtual addresses of code and data are randomized, making it difficult for an attacker to succeed [16, 17]. Each byte of the program code can be scrambled in load time using pseudorandom numbers [18]. All these techniques have a significant performance overhead. The overhead can be reduced if static analysis is used to determine which parts of the code should be protected by dynamic detection [19].

Several researchers suggest intrusion detection by monitoring the system calls of a program [20-23]. If the system call sequence for a particular program deviates from a normal behavior, an intrusion is suggested. The normal program behavior is obtained either by profiling, or by encoding the specification of expected behavior using a special high-level specification language. If profiling is used, false positives may be generated when a rarely used region of the code is executed. A specification-based approach, on the other hand, is as error prone as the coding process itself. Finally, although a malicious code is very likely to encompass a system call, such as the system() command, an attacker may potentially devise an attack with the same call sequence as the vulnerable program, or inflict some damage even without system calls. Another profiling approach [24] suggests using the values of performance monitoring registers to verify whether the program deviates from its expected behavior.

Some of the performance overhead may be reduced with hardware support. Xu et al. propose an architectural support against the buffer overflow attack: a return address is saved on both the Secure Return Address Stack and on the "regular" stack [25]. An attack is detected if the two addresses do not match. Similar efforts expand this idea [26, 27]. 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 [28], but at the price of a significant increase in execution time.

Kirovski et al. propose the Secure Program Execution Framework for intrusion prevention [29]. The underlying idea is that the executable of a program can have different representations that produce the correct program behavior. Possible code transformations include instruction scheduling, basic block reordering, branch-type selection, register permutation, etc. During installation, a transformation-invariant (TI) hash value is calculated for each instruction block and is encrypted using a secret processor key. The encrypted hash value defines the transformation of the instruction block. During execution, the verifier component calculates the TI hash for every instruction block that is fetched after an instruction cache miss. It then encrypts the hashed value, and verifies whether the obtained transformation is equal to the actual code. If there is no match, an abort signal is sent to the processor. This solution successfully prevents execution of injected code, but at the cost of relatively significant performance overhead. It must be customized for different platforms and a particular instruction set.

An 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 [30]. 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.

Signatures of instruction blocks of various granularity are frequently used in fault-tolerant computing [31]. Joseph and Avizienis proposed the idea of a virus protection technique using an extended Program Flow Monitor -- an error detection mechanism that verifies the signature of the sequence of instructions without any branch instructions [32]. However, the paper does not include any implementation details or evaluation.

3. PROCESSOR EXTENSIONS FOR TRUSTED INSTRUCTION EXECUTION

All three approaches introduce relatively modest changes in processor organization. We will first describe the details of the SIGT mechanism, and then the differences between the SIGT and the other two proposed processor extensions.

3. 1. SIGT Implementation

Processor and Memory Segment Modifications. A secure processor includes a dedicated resource for signature verification, the Instruction Block Signature Verification Unit (IBSVU) (Figure 1), dedicated registers, and additional control logic. The processor also includes the Instruction Block Signature Table (IBST). The IBST is a cache-like structure that keeps relevant information about the most recently needed instruction block signatures. The signature information for all instruction blocks is stored in the IBST_M table in main memory.

Compilation and Program Installation. The program compilation process generates a list of all basic blocks in the code and appends it to the executable (Figure 1). Disassembling can extract this list from the executable with more than 99% accuracy [11], but some basic blocks may not be easily recognized. During secure installation, the code is augmented with encrypted instruction block signatures, where a signature is a function of the instruction words in the block. Although different functions can be used for the signature, we propose the use of a multiple input signature register (MISR). The signature of a block is calculated with the instruction words as consecutive inputs to the MISR. The calculated signature is then encrypted using a processor key hidden in hardware, which is unique to each processor, and some symmetric secure encryption algorithm such as AES (Advanced Encryption Standard). For each new block, the MISR is initialized to the same value. The MISR linear feedback coefficients are also based on the hardware encryption key. Signatures are stored in a separate code segment: the instruction block signature table in memory (IBST_M). Each entry in the IBST_M includes the IB.SA (Instruction Block Starting Address Offset) and IB.S (Instruction Block Signature) fields. The IB.SA is the offset of the address of the first instruction in the basic block from the beginning of the code. This field is used as a key for accessing the IBST_M.

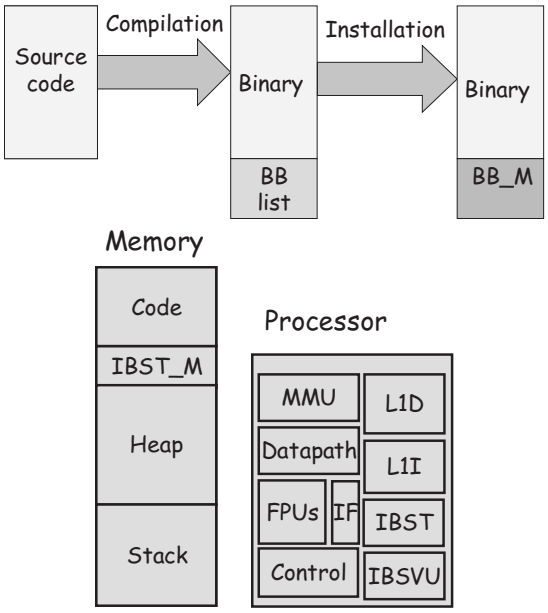

Figure 1 SIGT: Compilation and installation, and memory and processor modifications

Legend: MMU – Memory Management Unit, IF – Instruction Fetch Unit, FPUs – Floating Point Unit(s), Control – Control Unit, L1D – Level 1 Data Cache, L1I – Level1 Instruction Cache, IBST – Instruction Block Signature Table, IBST_M – Instruction Block Signature Table in Memory, IBSVU – Instruction Block Signature Verification Unit.

The library code deserves a special note. If a static library is used, only the necessary functions are linked with the rest of the application into one executable file. Basic block signatures are calculated for that file, so the signature table includes signatures of basic blocks in the used library functions. Each dynamically linked library (DLL) has its own signature table, and the pointers

to that table can be loaded at load time, so all code can be safely verified.

Program Loading. Signatures can be decrypted during program loading from a hard disk to memory, or when a particular signature is fetched from memory during program execution. If the IBST_M is decrypted at load time, the decryption overhead is concentrated at the beginning of program execution. This approach also avoids re-decryption of the same signature when that signature must be fetched in the IBST from the IBST_M more than once. A memory region with the signatures must be protected, so it is accessible only by the IBSVU. Otherwise, an attacker might inject malicious basic blocks and change the corresponding signatures.

The alternative approach for signature decryption is to decrypt signatures in the run time, possibly by a dedicated hardware resource. Run-time decryption does not require protected memory, since signatures in the memory are encrypted.

A subset of signatures can be preloaded into the processor's IBST, and that subset can be chosen in various ways: by spatial locality, by applying profiling information, or randomly. Another option is not to preload the IBST, but to fill it dynamically, just like a regular cache structure.

Program Execution. To reduce the number of verifications, we can optionally perform verification only for the last basic block in an instruction stream (a dynamic basic block), since any injected code will most likely change the control flow. Without loss of generality, we consider a case where the size of all instructions is 4 bytes, and the mechanism verifies the signature of the last basic block in an instruction stream. Figure 2 shows a block scheme of program flow monitoring.

When the instruction decoder recognizes the end of a basic block, i.e., a control-flow changing instruction, it asserts the signal *NewIB* (New Instruction Block) for instruction that follows. The offset of the new basic block from the beginning of the code is calculated by deducting the value in the PC register from the value stored in the SA (program Starting Address) register, and stored in the CB.SA (Current Block Starting Address) register. The signature for the current basic block is calculated by using values of instruction words stored in the IR register, and the same MISR coefficients that are used for signature generation. The MISR is reset at the start of each new basic block. The current basic block signature is stored in the CB.S (Current Block Signature) register.

The IBSVU needs to verify a basic block signature only if that block caused at least one instruction cache miss (signal *ICacheMiss*), and when that basic block was the last block in an instruction stream (signal *NewStream*). The end of the current instruction stream is detected by comparison of the PC (Program Counter) and PPC (Previous Program Counter) registers, where the PPC is an additional register saving the value of the previous PC.

If both the *NewStream* and *ICacheMiss* signals are asserted, current values of the CB registers (CB.SA and CB.S) are transferred to the corresponding LB (Last Block) registers in the IBSVU.

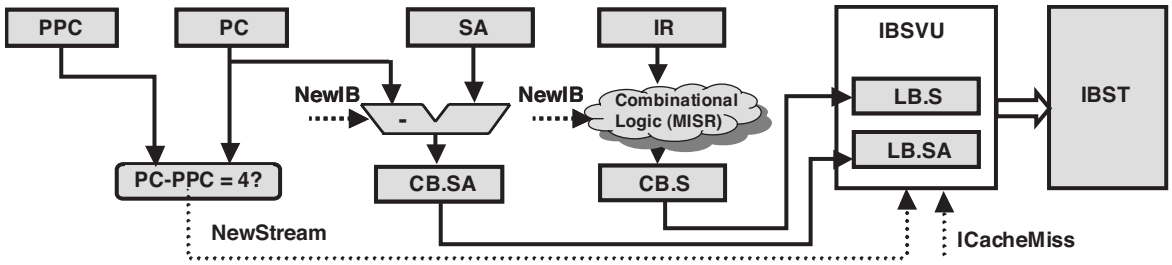

Figure 2 SIGT program flow monitoring with fixed instruction size

Legend: PC – Program Counter, PPC – Previous Program Counter, IR – Instruction Register, SA- Starting Address, IBSVU - Instruction Block Signature Verification Unit, CB.S/LB.S – Current Block/Last Block Signature Register, CB.SA/LB.SA Current Block/Last Block Starting Address Offset. Control signals are shown in dotted lines: *ICacheMiss* – indication of an instruction cache miss during basic block execution, *NewIB* – control signal from the decoder, indicating the beginning of a new basic block, *NewStream* – indication of the beginning of the new stream.

The CB registers then continue to capture the relevant information of the currently executed basic block, while concurrently the IBSVU is verifying the signature of the last basic block in the previous stream by comparing it to the corresponding data in the IBST. The signature can be captured in parallel with the execution pipeline stage for in-order execution, and after the decode stage for out-of-order execution.

The IBST lookup results in a miss or a hit. In the case of a signature hit the executed instruction stream has no malicious code. A signature miss can be an infrequently executed basic block or injected code, so the IBST_M must be searched for the signature with the matching starting address offset. Since an IBST_M does not change for a given program, the secure installation process may find a near-perfect hash function for a particular application, or choose the most suitable hash function from a predefined set of functions. The information about the chosen hash function can be kept in the program header in an encrypted form. If there is a corresponding entry in the IBST_M and there is a signature match, the instruction stream is not injected: the execution continues as usual, and the IBST is updated. Otherwise, it means that a malicious instruction stream has been executed, so the IBSVU traps the operating system. The operating system then halts the program execution and audits the intrusion event.

3. 2. SIGE implementation

Processor and Memory Segment Modifications. The SIGE requires less complex hardware resources than the SIGT, since there is no IBST table. Since the signatures are embedded in the code, there is no additional memory segment for signatures.

Compilation and Program Installation. Similarly to the SIGT, during secure installation signatures are generated using a list of basic blocks prepared by the compiler. The signatures are embedded in the code, with each signature placed before the first instruction in the corresponding instruction block (Figure 3). The instruction decoder must be able to tell the difference between the 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 the following instruction is the signature.

Program Loading. Signatures are loaded in main memory together with the code, and decrypted when they are fetched from main memory to the cache.

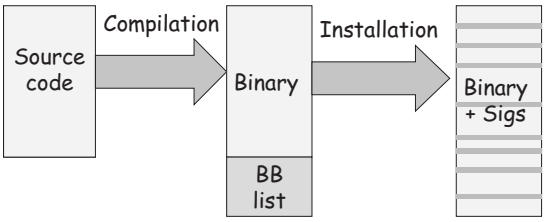

Figure 3 SIGE compilation and Installation

Program Execution. The verification process is similar to the SIGT: when current basic block is the last one in an instruction stream and caused at least one instruction cache miss, the signature embedded in the block is compared to the signature calculated during basic block execution.

3. 3. SIGC implementation

Processor and Memory Segment Modifications. In the SIGC, the IBSVU is a part of the cache controller (Figure 5), and the processor requires no changes. The organization of memory segments is not modified either.

Compilation and Program Installation. The SIGC does not require any changes in the compilation process, so it is even more suitable for legacy code. The signatures are calculated, encrypted, and inserted in the code during program installation, for each instruction block that corresponds to the size of the cache block in a given architecture. If the last instruction block is shorter than a cache block, it is padded with randomly chosen instructions that do not change the state of the processor. Since all instruction blocks in the SIGC have the same size, there is no need for the change of the ISA.

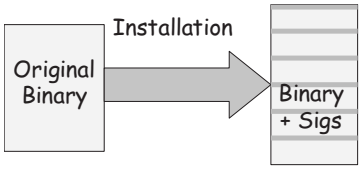

Figure 4 SIGC Installation

Program Loading. A signature of an instruction block is decrypted when that block is fetched from main memory.

Program Execution. On a cache miss, the corresponding block is fetched from the memory; the signature is stored in the signature register, and the instruction block is stored in the cache without its signature. Hence, there will be no additional cache

misses due to embedded signatures as in the SIGE, although we can expect a slight increase in the number of page misses. As an encrypted signature precedes its instruction block, it is fetched first and decrypted concurrently with the transfer of the rest of the block from memory. The calculation of the actual signature using a MISR is also overlapped with the memory transfer, so the whole verification process can be done in the background. The SIGC includes a simple mechanism for translation of program addresses into domain of addresses with embedded signatures.

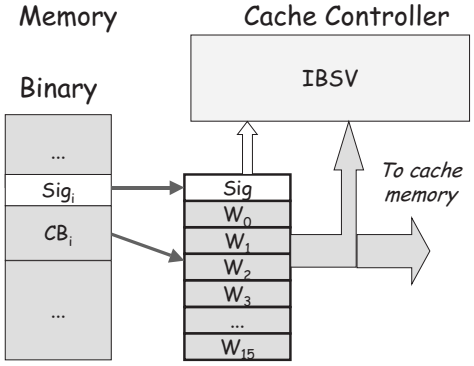

Figure 5 SIGC mechanism

4. PRELIMINARY RESULTS

Preliminary evaluation has been performed to assess the performance overhead. Due to the ever-increasing processor-memory speed gap, the memory access overhead will be the predominant overhead component. To assess this overhead, we measured the number of IBST misses for the SIGT, and the number of additional instruction cache misses for the SIGE. The miss rates are measured using an originally developed functional trace-driven simulator and SPEC CPU2000 traces collected for Alpha architecture [33]. The overhead of the SIGC mechanism is evaluated using a modified SimpleScalar simulator [34]: we take into account the latency due to additional memory accesses for signature fetching and to TLB misses due to address translation.

We use 10 integer (INT) and 12 floating-point (FP) applications (Table 1). Each application is run in two segments for the reference data input: the first two billion instructions (F2B), and the two billion instructions after skipping 50 billion (M2B). The IBST_M and the code with embedded signatures are generated using the complete code, and not only the executed basic blocks. The instruction cache size is fixed for all experiments, with 64B lines, 4 ways, 128 sets, and the least recently used replacement policy (LRU).

Table 1 shows the number of unique basic blocks executed in each traced segment (F2B/M2B), executed in complete benchmark execution (All), and identified in the code (Code). The results in the table indicate feasibility of the SIGT, as the number of executed basic blocks is relatively small and as they exhibit strong temporal locality. The code expansion for all three mechanisms can be calculated as:

$$Expansion = CodeSize / (NumBlocks \times SignatureSize),$$

where *NumBlocks* is the number of basic blocks in the code for SIGT and SIGE and the code size divided by the size of a cache block for SIGC. Our preliminary evaluation does not include effects of the increased code size in main memory. In the future we plan to simulate context switches between several applications and additional hard disk I/O due to the increased code size.

Table 1 Unique basic blocks and code size

	Unique basic blocks				Code Size [B]
	F2B	M2B	All	Code	
164.gzip	872	327	1480	8660	212992
176.gcc	29133	25777	32493	98478	1990656
181.mcf	981	327	1399	7401	163840
186.crafty	4161	1692	4801	17761	442368
197.parser	4193	4145	5597	14663	319488
252.eon	3885	675	4298	24285	794624
253.perlbmk	10425	7542	12290	43294	876544
254.gap	3580	542	3740	47365	933888
255.vortex	8086	3823	11765	33336	819200
300.twolf	2842	1195	5425	17931	450560
FP					
168.wupwise	2132	312	2435	32989	819200
171.swim	2268	793	2510	32759	819200
172.mgrid	1909	1082	2140	32312	802816
177.mesa	2177	763	2452	33757	917504
178.galgel	2518	166	5797	41805	1048576
179.art	549	502	1222	9600	237568
183.equake	668	395	1629	9436	253952
188.ammp	1100	566	2032	19917	385024
189.lucas	1318	458	1833	33246	851968
191.fma3d	2447	1082	5820	59790	1867776
200.sixtrack	4325	144	7859	61938	2596864
301.appsi	2439	636	3867	35393	1114112

The SIGT and SIGE experiments use a 32-bit signature. An attacker may have knowledge only about the program code, and not about the signatures, so it is very difficult to discover the MISR function by cryptanalysis [28]. For example, a brute force buffer overflow attack would need to overflow the buffer up to 2^{32} times to find a basic block with a signature that is accepted by the system. Nevertheless, if more security is needed, we may use longer signatures or a different MISR function for each installed program, with the corresponding MISR coefficients stored in the program header in an encrypted form. Since SIGC is simulated in more details, we use a more realistic signature size of 128 bits.

Figure 6 shows the number of IBST misses per one million executed instructions (SIGT). Each IBST miss causes additional memory accesses for the IBST_M search. The simulated IBST is filled dynamically, has LRU replacement policy, and two ways; one IBST block contains one signature. We simulated the IBST that can hold 32, 64, 128, 256 and 512 signatures. Only three INT applications in the F2B segment have over 1000 misses per 1M instructions for all simulated sizes -- *255.vortex*, *176.gcc*, and *253.perlbmk* -- and of the FP applications only *191.fma3d* in the F2B segment has over 1000 misses, and only for the smallest simulated sizes. On average, the number of IBST misses is smaller in the M2B segment than in the F2B, due to the smaller number of cache misses in the M2B, when most applications enter the main program loop. The results indicate that a very small IBST size is enough for most simulated applications. We also evaluated the influence of IBST associativity to the number of misses for an IBST with 128 entries, and direct mapped organization, 2, 4, and 8 ways. Most applications do not significantly benefit from more than two ways.

In the SIGE the signatures are fetched from memory into the instruction cache together with the regular instructions, so there are no extra memory accesses for signature verification, but the overall number of cache misses increases. To assess the SIGE potential, we compared the number of instruction cache misses per one million instructions for the original code and for the code with embedded signatures (Figure 7).

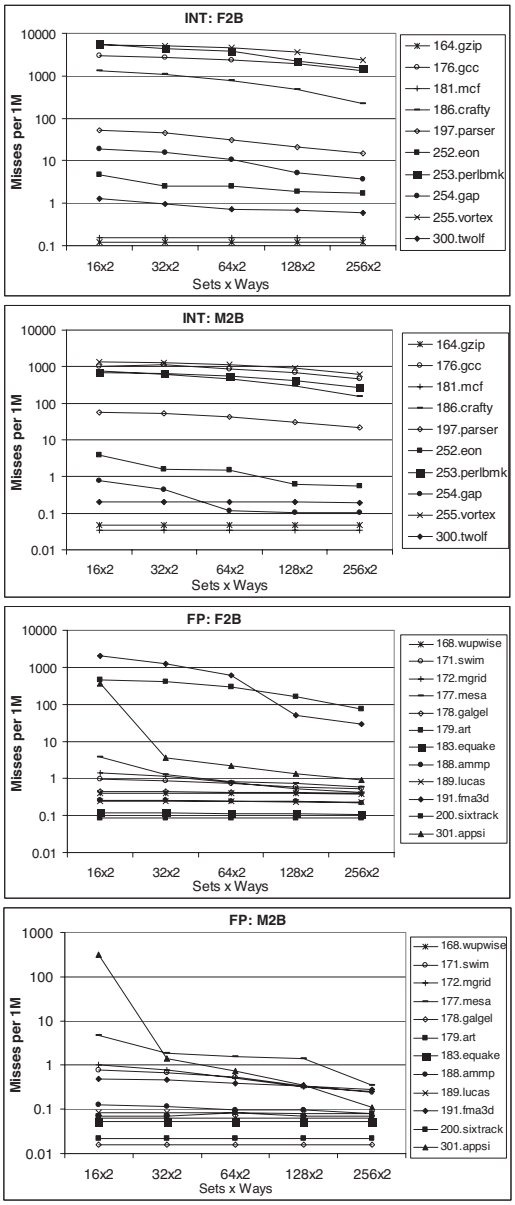

Figure 6 IBST misses per 1M instructions

Since most applications have relatively few instruction cache misses, the SIGE should not significantly influence overall program performance. For one application in the F2B segment, *183.equake*, the number of cache misses is even reduced, due to the better alignment of some portions of the code. However, for some applications the increase in the number of cache misses can be considerable: for example, for *252.eon* this number increases for one

order of magnitude, from about 300 to about 4000 cache misses per one million instructions.

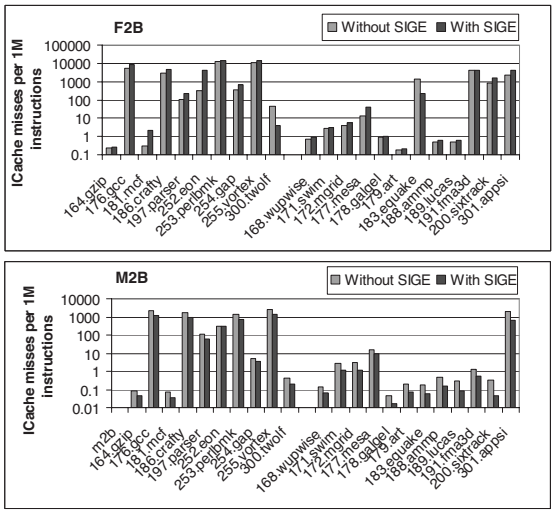

Figure 7 Instruction cache misses for SIGE vs. code without embedded signatures

The increase in the number of cache misses is avoided with the SIGC, since signatures are stripped before an instruction block is stored in the cache. The overhead of the SIGC is evaluated by comparing IPC (instructions per cycle) measure for original code and the code with embedded signatures. All sim-outorder simulator parameters except cache sizes have default values. We simulated two instruction cache L1 sizes: 32K (64B line, 4 ways, LRU) and 4K (32B line, direct mapped). The L1 data cache is the same as the instruction cache, and there is no L2 cache. Figure 8 shows the results for the F2B segment. Since a signature is stored at the beginning of an instruction block, we assume that signature decryption can be overlapped with memory accesses for the rest of the block. For simulated architectures the SIGC does not significantly change the IPC: up to 0.075 for 4K cache, and up to 0.056 for 32K cache.

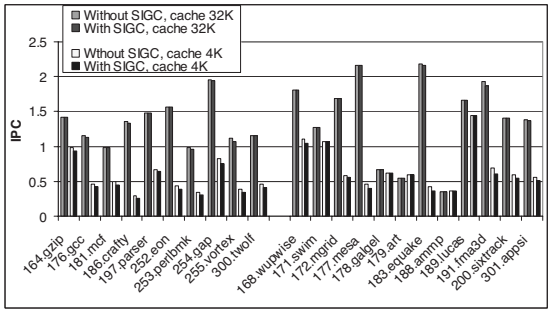

Figure 8 IPC increase with SIGC

5. CONCLUSION

The contributions of this paper are as follows:

- Proposal of a cost-effective architecture for trusted program execution based on the verification of the instruction block signatures. We believe that processor extensions for verification of instruction block signatures can be an efficient and inexpensive defense against attacks injecting malicious code.

- Three implementations of the proposed extensions, with signatures stored in the separate code section (SIGT), embedded in the code (SIGE), embedded in the code but not stored in the cache (SIGC). The proposed trusted execution mechanism can be applied to other purposes, such as fault-tolerant execution, virus protection, and protection from software tampering.

- Initial performance evaluation, based on the functional simulation of execution of SPEC CPU2000 benchmarks. The results suggest that the proposed implementations do not impose significant burden on the overall performance. In the SIGT the number of IBST misses is relatively small, even for the smallest simulated IBST: for 16 sets and 2 ways, the number of IBST misses per one million instructions varies from less than 1 to 5700. In the SIGC, simulated with cycle-by-cycle simulator, the increase in the IPC ranges from 0 to 0.075.

Future work will include cycle-by-cycle simulation in the design space of current and future microprocessors, the effects of signature decryption and context switching, and power analysis. Different IBST_M access functions should also be explored, as well as whether profiling information can reduce the number of IBST misses. We also plan to evaluate a variant of the SIGT, where the size of an instruction block is equal to the cache block size.

REFERENCES

[1] J. Wilander and M. Kamkar, "A Comparison of Publicly Available Tools for Dynamic Buffer Overflow Prevention," Proceedings of the 10th Network and Distributed System Security Symposium, San Diego, California, 2003, pp. 149--162.

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

[3] M. Milenkovic, A. Milenkovic, and E. Jovanov, "A Framework For Trusted Instruction Execution Via Basic Block Signature Verification," 42nd Annual ACM Southeast Conference, Huntsville, Alabama, 2004.

[4] W. Landi, "Undecidability of static analysis," *ACM Letters on Programming Languages and Systems (LOPLAS)*, vol. 1, pp. 323 - 337, 1992.

[5] D. Wagner, J. S. Foster, E. A. Brewer, and A. Aiken, "A first step towards automated detection of buffer overrun vulnerabilities," Network and Distributed System Security Symposium (NDCS), San Diego, CA, 2000.
[6] D. Larochelle and D. Evans, "Statically detecting likely buffer overflow vulnerabilities," Proceedings of the 10th USENIX Security Symposium, Washington, D.C, 2001, pp. 177-189.
[7] N. Dor, M. Rodeh, and M. Sagiv, "CSSV: Towards a Realistic Tool for Statically Detecting All Buffer Overflows in C," ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, San Diego, California, USA., 2003, pp. 155 - 167.
[8] 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 overflowattacks," 7th USENIX Security Conference, San Antonio, Texas, 1998, pp. 63-78.
[9] K.-s. Lhee and S. J. Chapin, "Type-Assisted Dynamic Buffer Overflow Detection," 11th USENIX Security Symposium, 2002, pp. 81 - 88.
[10] C. Fetzer and Z. Xiao, "Detecting heap smashing attacks through fault containment wrappers," 20th IEEE Symposium on Reliable Distributed Systems, New Orleans, LA, USA, 2001, pp. 80-89.
[11] M. Prasad and T.-c. Chiueh, "A Binary Rewriting Defense Against Stack-based Buffer Overflow Attacks," Usenix Annual Technical Conference, San Antonio, TX, 2003, pp. 211-224.
[12] J. L. Steffen, "Adding run-time checking to the portable C compiler," *Software—Practice & Experience*, vol. 22, pp. 305 - 316, 1992.
[13] T. Jim, G. Morrisett, D. Grossman, M. Hicks, J. Cheney, and Y. Wang, "Cyclone: A Safe Dialect of C," USENIX Annual Technical Conference, Monterey, CA, 2002, pp. 275-288.
[14] M. Bernaschi, E. Gabrielli, and L. V. Mancini, "Operating system enhancements to prevent the misuse of system calls," Conference on Computer and Communications Security, Athens, Greece, 2000, pp. 174 - 183.
[15] "http://pax.grsecurity.net/", (February 2004).
[16] P. Busser, "Memory Protection with PaX and the Stack Smashing Protector: Breaking out Peace," *Linux Magazine*, pp. 36-39, 2004.
[17] S. Bhatkar, D. C. DuVarney, and R. Sekar, "Address Obfuscation: An Approach to Combat Buffer Overflows, Format-String Attacks, and More," 12th USENIX Security Symposium, Washington, DC, 2003, pp.
[18] 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," 10th ACM Conference on Computer and Communication Security, Washington, DC, USA, 2003, pp. 281 - 289.
[19] S. H. Yong and S. Horwitz, "Protecting C Programs from Attacks via Invalid Pointer Dereferences," 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Helsinki, Finland, 2003, pp. 307-316.
[20] R. Sekar, T. Bowen, and M. Segal, "On Preventing Intrusions by Process Behavior Monitoring," 8th USENIX Security Symposium, Washington, DC, 1999, pp. 29-40.
[21] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," IEEE Symposium on Security and Privacy, Oakland, CA, 1999, pp. 133-145.
[22] I. Sato, Y. Okazaki, and S. Goto, "An Improved Intrusion Detection Method Based on Process Profiling," *IPSJ Journal*, vol. 43, pp. 3316-3326, 2002.
[23] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion Detection using Sequences of System Calls," *Journal of Computer Security*, vol. 6, pp. 151-180, 1998.
[24] D. L. Oppenheimer and M. R. Martonosi, "Performance Signatures: A Mechanism for Intrusion Detection," Proceedings of the 1997 IEEE Information Survivability Workshop, San Diego, California, 1997.
[25] J. Xu, Z. Kalbarczyk, S. Patel, and R. K. Iyer, "Architecture Support for Defending Against Buffer Overflow Attacks," Workshop on Evaluating and Architecting System dependability (EASY), San Jose, California, 2002.
[26] R. B. Lee, D. K. Karig, J. P. McGregor, and Z. Shi, "Enlisting Hardware Architecture to Thwart Malicious Code Injection," Security in Pervasive Computing, Boppard, Germany, 2003, pp. 237-252.
[27] 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.
[28] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," 10th ACM Conference on Computer and Communication Security, Washington, DC, USA, 2003, pp. 272 - 280.
[29] D. Kirovski, M. Drinic, and M. Potkonjak, "Enabling trusted software integrity," Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X), San Jose, California, 2002, pp. 108 - 120.
[30] G. E. Suh, J. W. Lee, and S. Devadas, "Secure Program Execution via Dynamic Information Flow Tracking," ASPLOS-XI, Boston, MA, 2004.
[31] A. Mahmood and E. J. McCluskey, "Concurrent error detection using watchdog processors-a survey," *IEEE Transactions on Computers*, vol. 37, pp. 160 - 174, 1988.
[32] M. K. Joseph and A. Avizienis, "A fault tolerance approach to computer viruses," Proceedings of the 1988 IEEE Symposium on Security and Privacy, Oakland, California, USA, 1988, pp. 52-58.
[33] A. Milenkovic and M. Milenkovic, "Exploiting Streams in Instruction and Data Address Trace Compression," Proceedings of IEEE 6th Annual Workshop on Workload Characterization, Austin, TX, 2003, pp. 99-107.
[34] D. Burger and T. Austin, "The SimpleScalar Tool Set Version 2.0," University of Wisconsin, Technical Report CS-TR-97-1342, 1997.