A Low Overhead Hardware Technique for Software Integrity and Confidentiality

Austin Rogers[†], Milena Milenković[§], Aleksandar Milenković[¶]
[†] *Dynetics, Huntsville, Alabama*; [§] *IBM, Austin, Texas*
[¶] *ECE Department, The University of Alabama in Huntsville*
Email: [¶] *milenka@ece.uah.edu*

Abstract

Software integrity and confidentiality play a central role in making embedded computer systems resilient to various malicious actions, such as software attacks; probing and tampering with buses, memory, and I/O devices; and reverse engineering. In this paper we describe an efficient hardware mechanism that protects software integrity and guarantees software confidentiality. To provide software integrity, each instruction block is signed during program installation with a cryptographically secure signature. The signatures embedded in the code are verified during program execution. Software confidentiality is provided by encrypting instruction blocks. To achieve low performance overhead, the proposed mechanism combines several architectural enhancements: a variation of one-time-pad encryption, parallelizable signatures, and conditional execution of unverified instructions. A relatively high memory overhead due to embedded signatures can be reduced by protecting multiple instruction blocks with one signature, with minimal effects on complexity and performance overhead.

1. Introduction

With current trends toward computer systems' ubiquitous accessibility, connectivity, diversification, and proliferation, computer security has become a critical issue in computer system design and operation. A need for computing devices resilient to various attacks is further underscored by a growing number of software vulnerability exploits, as well as by ever-increasing system complexity and tightening time-to-market constraints leaving less time for system testing.

Depending on the nature of the threat, computer security encompasses three components: confidentiality, integrity, and availability. Confidentiality is violated whenever information is disclosed to unauthorized entities (humans, programs, computer systems). Integrity is violated whenever information (code or data) is altered by an unauthorized entity. Availability is violated whenever an attacker succeeds in denying services to legitimate users.

Software integrity is ensured if the computer system can detect any unauthorized code and prevent its execution. Software confidentiality is ensured if an adversary cannot read binaries, thus preventing software duplication through reverse engineering and intellectual property theft. Attacks against software integrity are typically launched across the network by exploiting known software vulnerabilities (e.g., buffer overflow or unsafe format string). In embedded systems, adversaries can easily get physical access to the system and can probe buses and tamper with code and data in main memory, hard disk, and I/O devices. As a result, the computer system becomes vulnerable to many attacks, such as spoofing, splicing, and replay. In a spoofing attack, an adversary intercepts a request for an instruction block (I-block) and returns a block with malicious code instead. In a splicing attack, an adversary intercepts a request for an I-block and returns another valid I-block, but not the one that has been requested. In a replay attack, an adversary returns a stale copy of the requested data block.

The multitude of software attacks has prompted the development of a large number of predominantly software-based counter-measures. Static software techniques rely on formal analysis and/or programmers' annotations to detect security flaws in the code, and then leave it to the programmers to correct these flaws. Dynamic software techniques augment the original code or operating system to detect malicious attacks at runtime and to terminate attacked programs. Software techniques are unlikely to counter all attacks by themselves, as they lack generality, suffer from false-positives and false-negatives, and often induce prohibitive overhead in performance and power consumption. On the other hand, a further increase in the number of transistors on a single chip will enable integrated hardware support for functions that so far have been restricted to the software domain.

Several recent research efforts propose hardware-assisted techniques to prevent execution of unauthorized code [1-4]. These techniques promise

1-4244-1258-7/07/$25.00 ©2007 IEEE

higher security with relatively modest overheads in performance or energy. However, the existing techniques often fail to counter all attacks, induce prohibitive overheads, or their evaluation does not explore the implications of various implementation choices. Several general purpose processors now include security extensions, for example, IBM's Secure Blue for PowerPC, and AMD's Athlon-64 and Intel's Itanium buffer overflow protection.

In this paper we propose a hardware-supported technique for software integrity and confidentiality in embedded platforms that provides a maximum level of security at minimal cost, power overhead, and performance loss. Software integrity is ensured by signing I-blocks by a parallelizable cryptographically secure signature. During program execution, the signature is recalculated from instructions in the I-block and compared to the signature embedded in the executable code. If the two values do not match, the program cannot be trusted, and it is terminated. Software confidentiality is provided by encrypting instruction blocks using a variation of the one-time-pad technique (OTP).

The proposed technique overcomes shortcomings of the previously proposed techniques for software integrity. It relies on cryptographically strong but parallelizable signatures and supports code confidentiality. We also propose a novel cost-effective signature verification implementation: unverified instructions can be executed but they cannot commit their results until the verification is done. This implementation almost completely hides the overhead of I-block verification. A relatively high memory overhead because of embedded signatures can be reduced by protecting multiple I-cache blocks with one signature.

The experimental analysis encompasses evaluation of performance, power, and memory overheads. The results of cycle-accurate simulations for multiple machine models indicate negligible overhead in performance: from ~4% for platforms with extremely small instruction caches to almost 0% for platforms with relatively large instruction caches. This result is achieved with low overhead in energy consumed (less than 5%).

The rest of the paper is organized as follows. Section 2 describes the proposed architecture for software integrity and confidentiality and discusses its implementation. Section 3 details the experimental environment used for performance analysis. Section 4 shows results of the experimental evaluation. Section 5 surveys hardware-assisted techniques for software integrity and confidentiality and addresses potential weaknesses of the existing techniques, and Section 6 concludes the paper.

2. Architecture for Software Integrity & Confidentiality

The proposed technique encompasses the following components: secure program installation, program loading, and program execution (Figure 1). Depending on the required level of protection, a program can run in an unprotected mode, a software integrity only mode (SIOM), or a software integrity and confidentiality mode (SICM). Information about the selected security mode is stored in the program header during secure installation.

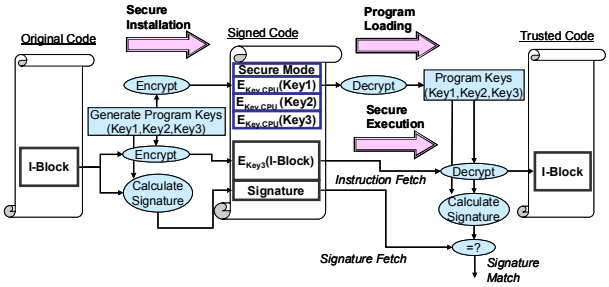

Figure 1. Mechanism for Software Integrity and Confidentiality.

2.1. Secure Program Installation

Secure program installation encompasses key generation, signature generation, and code encryption. Secure program installation is performed in a special installation mode not interruptible by other processes, similar to the installation mode described by Kirovski *et al.* [2].

Key generation. Depending on the selected security mode, a program requires zero, 2 or 3 keys (Key1, Key2, Key3). The keys may be generated using thermal noise within the processor, or by physical unclonable functions [5]. These keys are encrypted with a secret processor key (Key.CPU), and stored in the program header (see blue boxes in Figure 1). Plaintext keys must never leave the processor chip, so key generation and encryption are done using only on-chip resources.

Signature calculation. To ensure software integrity, each I-block is signed with a signature during the secure installation process and signatures are embedded in the executable code. An I-block signature is a cryptographic function of the following: (a) the starting virtual address of the I-block or offset from the beginning of the code section, (b) unique program keys created during secure installation, and (c) instruction words in the I-block. Using the starting virtual address prevents any splicing attacks, because a legitimately signed I-block from a different address will result in a

different signature than that calculated with the requested address. Using unique program keys should prevent execution of any unauthorized code, regardless of whether it has been injected by a software attack or inserted by a physical attack. It also prevents an adversary from replaying a valid instruction block of one program with a valid instruction block of another program residing at the same virtual address. Using instruction words is necessary to prevent any splicing or spoofing attacks.

Signatures are generated using the parallelizable MAC (PMAC) algorithm, developed by Black and Rogaway [6]. They prove PMAC secure and show that it approximates a random permutation. To illustrate the process of signature generation, we assume a 32-bit architecture, 32B I-blocks, 128-bit signatures appended as footers to I-blocks, and each I-block partitioned into two sub-blocks: ($I_{0:3}$), ($I_{4:7}$). For each sub-block SB_i ($i = 0, 1$), the signature generation is described in Eq. 1, where $Sig(SB_i)$ is sub-block signature, SP is a secure padding function, $A(SB_i)$ is the starting virtual address of the sub-block, and Key1 and Key2 are secure program keys. The I-block signature S is an XOR function of all sub-block signatures (Eq. 2). Signatures prevent tampering with the code, but the code can still be inspected by an adversary. To provide software confidentiality, we can expand this scheme with software encryption.

Eq. 1 $$Sig(SB_i) = AES_{KEY2}[(I_{4i:4i+3}) \, xor \, AES_{KEY1}(SP(A(SB_i)))]$$

Eq. 2 $$S = Sig(SB_0) \, xor \, Sig(SB_1) \, .$$

Software encryption. Software encryption should provide a high level of security, yet it should not cause significant delays in the critical path during signature verification and software decryption processes. In order to satisfy these requirements, we adopt an OTP-like encryption scheme. Depending on the order in which we encrypt an instruction block and calculate its signature, there are three possible approaches known in cryptography as *encrypt&sign*, *encrypt, then sign*, and *sign, then encrypt* (StE). These three schemes differ in security strength which is still a matter of debate. However, for our implementation, all three schemes have similar hardware complexity and we decided to use the StE scheme.

In StE, the signature is calculated on plaintext instructions, as described in Eq. 1 and Eq. 2, and then both instructions and the signature are encrypted, as described in Eq. 3 and Eq. 4. We use Key3 for code encryption because it is recommended that authentication and encryption should not use the same keys [7].

Eq. 3 $$(C_{4i:4i+3}) = (I_{4i:4i+3}) \, xor \, AES_{KEY3}(SP(A(SB_i))) \, , \; i = 0, 1.$$

Eq. 4 $$eS = S \, xor \, AES_{KEY3}(SP(A(S)))$$

Security considerations. Even with a mechanism that protects software integrity, a skilled attacker can exploit software vulnerabilities to change the target of an indirect jump or return instructions to different existing code sections (so-called arc injection attacks). The SICM mode makes creation of meaningful arc injection attacks much more difficult, but it does not prevent them. Complete protection from such attacks may be provided by using a dedicated resource to store allowed targets of indirect jumps and a secure stack [8], or by using data encryption.

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

2.2. Program Loading and Execution

Program loading. Unique program keys are loaded from the program header into dedicated processor registers. The program keys are decrypted using the hidden processor key (Key.CPU) and can only be accessed using dedicated processor resources: the program key generation unit and an instruction block signature verification unit (IBSVU). On a context switch, these keys are encrypted before they leave the processor, and are stored in the process control block.

Secure program execution. When an instruction is fetched from memory, the integrity of the corresponding I-block needs to be verified. Consequently, the most suitable instruction block size is the cache line size of the lowest level of the instruction cache (the cache that is the closest to the memory) or some multiple thereof, or the size of the fetch buffer in systems without the cache. Without loss of generality, in the rest of this paper we focus on a system with separated data and instruction first level caches and no second level cache. The instruction cache (I-cache) is a read-only resource, so the integrity is guaranteed for instructions already in the I-cache. Hence, signatures only need to be verified on I-cache misses. Signatures are not stored in the I-cache and they are not visible to the processor core at the time of execution. To achieve this, an additional step is needed for address translation that maps the original code to the code with embedded signatures and potential page padding.

Signatures are verified in parallel with program execution using the IBSVU. Fetched instructions pass through a logic block that calculates a signature in the

same way it was generated during secure installation. This calculated signature *cS* is then compared to the one fetched from memory (*S*). If the two values match, the instruction block can be trusted; if the values differ, a trap to the operating system is asserted. The operating system then neutralizes the process whose code integrity cannot be verified and possibly audits the event. The process of runtime verification depends on security mode.

Software integrity only mode. The signature *cS* is calculated as described in Eq. 1 and Eq. 2. Figure 2 illustrates the process of signature calculation and runtime verification. The arrow running from top to bottom indicates the flow of time. Encryption of securely padded sub-block addresses (dark shaded boxes in Figure 3) can be initiated at the beginning of the memory cycle that fetches the required I-block (blocks marked with I) and its signature (blocks marked with S). In this way the AES block outputs will be ready to be XOR-ed with incoming instructions, assuming that the crypto latency is less than the memory access time.

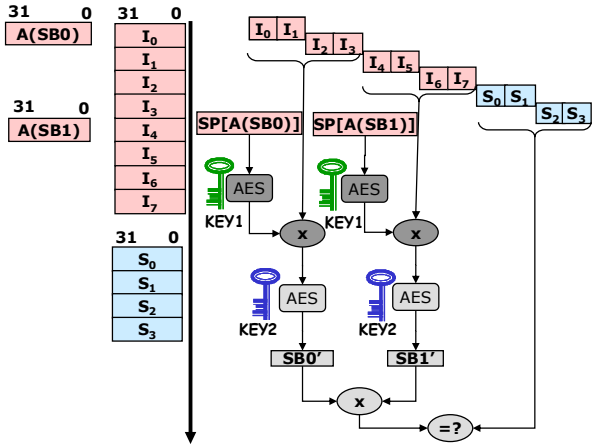

Figure 2. Runtime signature verification in SIOM mode.

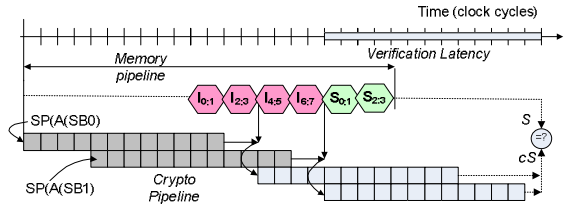

Figure 3. Memory and crypto pipeline for PMAC-based runtime signature verification.

A simple cipher implementation would use the CBC-MAC [9]. Assuming 32B I-blocks, 128-bit signatures, 12 clock cycle AES cipher implementation, 64-bit data bus, and 12/2 memory latency (12 clock cycles for the first chunk, 2 clock cycles for each thereafter), the verification process will be completed in 25 clock cycles after the last instruction is fetched. However, if the cipher is replaced with a PMAC, the instruction verification process will be completed in only 13 clock cycles after the last instruction is fetched (Figure 3).

Software integrity and confidentiality mode. The fetched ciphertext is decrypted, producing the original I-block with minimal delay - one XOR operation, since the AES encryption of virtual addresses is overlapped with memory latency (Eq. 5). The signature *cS* is calculated from decrypted instructions. The signature fetched from memory is also decrypted as described in Eq. 6.

Eq. 5 $(I_{4i:4i+3}) = (C_{4i:4i+3})\ xor\ AES_{KEY3}(SP(A(SB_i)))$, $i = 0,\ 1.$

Eq. 6 $S = eS\ xor\ AES_{KEY3}(SP(A(eS)))$

A naïve implementation of the proposed mechanism will not allow execution of instructions before they are verified (Wait-'til-Verified scheme – WtV). Consequently, each I-cache miss event will extend the processor wait time for the duration of I-block verification (13 clock cycles in the example). However, this verification latency can be alleviated if we keep track of instructions under verification (Run-before-Verification scheme – RbV). Instructions can start execution immediately after they are fetched, but they cannot commit before the whole block is verified. For in-order processors, an additional Instruction Verification Buffer resource is needed (Figure 4, top). This buffer is similar to the Sequential Authentication Buffer proposed by Shi *et al.* [10]. All instructions that belong to a block under verification as well as possibly verified instructions that follow the unverified instructions get an entry in this buffer. The instructions can commit when the IBSVU confirms that the I-block is secure (verified flag is set). It should be noted that this buffer is used only when I-cache misses occur, so a relatively small buffer will suffice. In out-of-order processors, this functionality can be implemented by adding a verified bit to the reorder buffer and not allowing instructions to retire until that bit is set. Shi and Lee [11] assert that schemes allowing unverified instructions to execute (but not commit until verification is complete) may allow sensitive data to be exposed by a malicious memory access instruction inserted in the block. However, this action would not violate the confidentiality of instructions, and the tampering would be evident after verification.

Memory overhead can be reduced by making the protected I-block size a multiple of I-cache block size. For instance, protecting two cache blocks with one signature will cut the memory overhead for the code section in half. On an I-cache miss, the entire protected block and its signature are fetched from memory. Once they are decrypted, the portion of the protected block

that is currently needed is placed into the I-cache. A naïve implementation would always discard the other portion of the block. Performance can be improved by placing I-block(s) after the I-block that caused an I-cache miss into an instruction opportunity buffer (IOB, Figure 4, bottom). The IOB is a FIFO queue with address tags searched in a fully associative manner. On future I-cache misses, the IOB is searched for the desired block. If there is an IOB hit, the data is taken from the IOB and placed into the I-cache, preventing a memory access and subsequent verification latency.

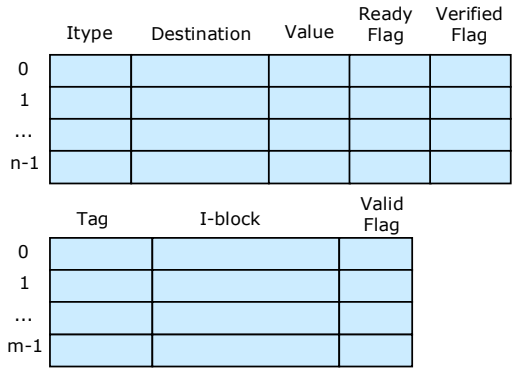

Figure 4. Instruction verification buffer (top) and instruction opportunity buffer (bottom).

3. Experimental Methodology

An ideal technique should provide an adequate level of software integrity and confidentiality, yet it should be easy to implement (minimal cost), should not impose significant delays in program execution or increased power consumption, and should not introduce significant memory overhead.

A qualitative assessment indicates relatively low hardware complexity of the proposed IBSVU – it includes a key generation unit, a pipelined AES cipher, temporary buffers, the instruction verification buffer (for RbV implementation), the instruction opportunity buffer (if multiple cache blocks are protected by a single signature), and control logic. The AES cipher, following the commercially available Cadence High Performance AES core, is comprised of approximately 57,000 logic gates [12]. The memory overhead is simply determined by comparing the sizes of the original code and the code with signatures. To emulate the secure installation process, we have developed a program that calculates signatures of instruction blocks in executable sections of programs in the ELF format, and modifies programs to include calculated signatures.

The performance overhead is evaluated using a modified SimpleScalar ARM simulator [13] that supports the proposed schemes. As a measure of performance, we use the number of clock cycles, normalized to the number of clock cycles for the base configuration (without signature verification). The benchmarks running on the base configuration exhibited the best performance for an I-cache line size of 32 bytes. Therefore, a 32 byte I-cache line size is used for evaluating the proposed integrity and confidentiality techniques, while the I-cache size varies between 1, 2, 4, and 8 KB. We assume a bus width of 64 bits. We use IOBs that are 25% of the size of the base I-cache, and protected block sizes of 64 bytes (twice the size of the I-cache block). The I-cache size is reduced by 25% to maintain a constant cache budget: it has the same number of sets and 3 instead of 4 ways.

The energy overhead is determined by comparing the total energy spent by the processor with the base configuration to the energy spent with signature verification. Energy is estimated using a modified Sim-Panalyzer ARM simulator [14], which models the effects of internal and external switching and leakage. For Sim-Panalyzer parameters related to power, we use values from a template file provided with Sim-Panalyzer. The operating frequency is 200MHz, the I/O supply voltage is 3.3V, and the internal logic power supply is 1V. The technology parameters correspond to the 0.18µm process. The AES cipher's power consumption is modeled as the power consumed by 57,000 gates.

As a workload for performance and power analysis, we use benchmarks from several benchmark suites for embedded systems (MiBench [15], MediaBench [16], BasicCrypt [17]). Since signature verification is done only at an I-cache miss, the benchmarks selected from these suites have a relatively high number of I-cache misses for at least one of the simulated cache sizes. Table 1 lists the benchmarks, their descriptions, the number of executed instructions, and the number of I-cache misses per 1000 executed instructions for the simulated cache sizes (1 – 8 KB).

Table 1. Benchmark descriptions and characteristics.

Benchmark	Description	Executed instr. [10^6]	I-cache misses per 1000 executed instr.			
			1K	2K	4K	8K
blowfish_enc	Blowfish encryption	544.0	33.8	5.1	0	0
cjpeg	JPEG compression	104.6	7.6	1.3	0.3	0.1
djpeg	JPEG decompression	23.4	11.9	5.5	1.3	0.3
ecdhb	Diffie-Hellman key exchange	122.5	28.5	8.5	2.9	0.1
ecelgencb	El-Gamal encryption	180.2	25.4	4.5	1.4	0.1
ispell	Spell checker	817.7	72.4	53	18.8	2.9
mpeg2_enc	MPEG2 compression	127.5	2.2	1.1	0.4	0.2
rijndael_enc	Rijndael encryption	307.9	110.2	108.3	69.5	10.3
stringsearch	String search	3.7	57.7	35	6.2	2.4

4. Results

Performance overhead. Figure 5 summarizes the total performance overhead of three analyzed architectures: WtV scheme with both CBC-MAC (CBC-MAC WtV) and PMAC (PMAC WtV) ciphers and RbV scheme with PMAC cipher (PMAC RbV). The bars in these figures indicate the average normalized execution time across all the benchmarks, while the candlestick lines indicate the minimum and maximum normalized execution times. They clearly show that the PMAC cipher with the RbV scheme yields a negligible performance overhead. Even with a very small 1KB I-cache, the total overhead with PMAC RbV is only 4% (min=0.4%, max=4.3%), compared to 63% (min=6%, max=80.6%) with PMAC WtV, and 93% (min=8.7%, max=117%) with CBC-MAC WtV. The total performance overhead decreases with an increase in the I-cache size, and it is almost non-existing with a relatively large 8 KB I-cache, compared to 5% with PMAC WtV, and 8% with CBC-MAC WtV. Longer memory latencies do not negatively influence the effectiveness of the proposed mechanism; on the contrary, the time spent in signature verification becomes relatively smaller in the total execution time, as illustrated by the results in Figure 5 bottom where memory latency is 24/2 clock cycles. Consequently, in the rest of this section we will present results for the modeled processor with 12/2 memory latency.

It should be noted that in our implementation the SICM and SIOM modes have the same overhead – an additional XOR gate delay in SICM mode implementation does not require a separate pipeline stage and the additional address decryption is fully overlapped with memory access.

Energy overhead. Figure 6 summarizes the total energy overhead of the three aforementioned architectures. The bars show the average normalized energy overhead across all the benchmarks, while the candlestick lines indicate the minimum and maximum normalized energy overhead. The normalized energy overhead roughly follows the performance overhead. The implementation with the PMAC cipher and the RbV scheme again introduces minimal power overhead.

Memory overhead. Memory overhead is a simple function of the size of signatures and the size of protected instruction blocks. The proposed signature is 16 bytes, and the size of a protected block corresponds to a multiple of the I-cache line size. Consequently, the executable section of an executable file increases 50% for 32-byte protected blocks, 25% for 64-byte, and 12.5% for 128-byte.

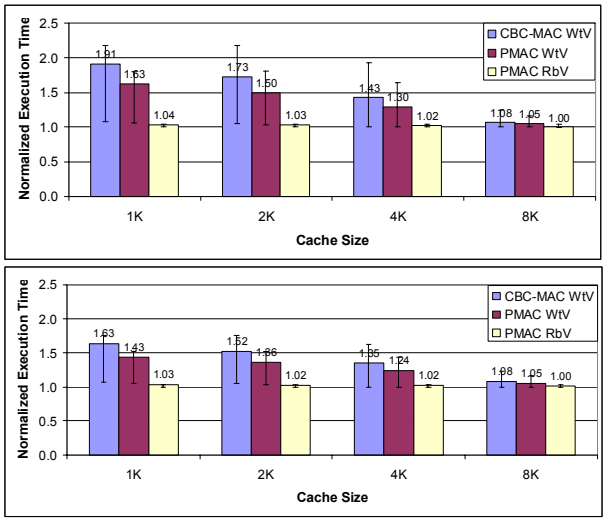

Figure 5. Performance overhead relative to the base configuration. Bus latency is 12/2 (top) and 24/2 (bottom).

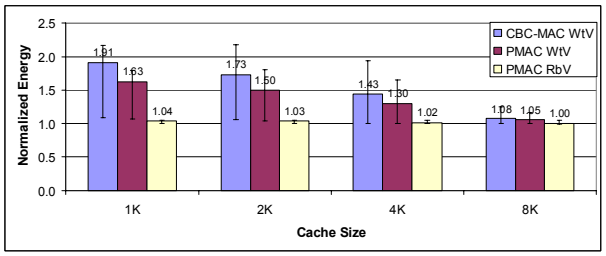

Figure 6. Energy overhead relative to the base configuration.

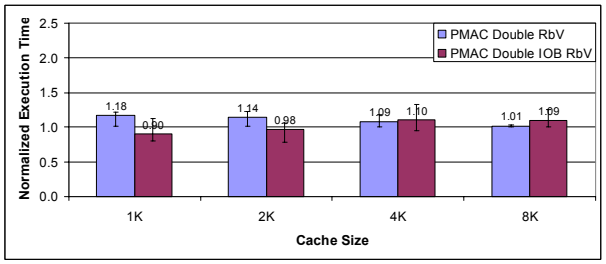

Figure 7. Performance overhead of reduced memory overhead implementations relative to the base configuration.

We consider two implementations which reduce memory overhead by using 64-byte protected blocks (i.e., twice the size of the I-cache blocks). Both implementations are based on the PMAC cipher and the RbV scheme. The first implementation naively discards currently unused blocks (PMAC Double RbV), while the second puts them into an IOB (PMAC Double IOB RbV). Figure 7 shows that the instruction opportunity buffer may even improve performance for smaller cache sizes, since the benefit of prefetching into the fully associative IOB may outweigh the loss of

one cache way. However, with larger caches the naïve implementation performs better. Several applications benefit a lot from an increase in cache size from 4 KB to 8 KB (*ispell*, *rijndael_enc*, and *stringsearch*, see Table 1), so the loss of one cache way significantly impacts performance and IOB hit rate is not sufficiently high to compensate for it.

5. Related Work

A number of hardware-assisted techniques have been proposed recently, differing in scope and complexity. A set of relatively simple techniques counters the most frequent stack smashing attacks by utilizing a secure hardware stack [18, 19] or through encryption of address pointers [20]. The use of untrustworthy data for jump target addresses can be prevented by tagging all data coming from untrustworthy channels [21, 22]; however, this approach requires relatively complex tracking of spurious data propagation and may produce false alarms. More comprehensive secure architectures that are directly related to this work include execute-only memory (XOM) [1], an architecture for protecting critical secrets in microprocessors [23], an architecture for memory integrity verification [24], a XOM-like architecture with fast one-time-pad encryption [3], an architecture for runtime verification of instruction block signatures [4], and a hardware/software platform for intrusion prevention [9].

Execute-only memory (XOM) strives to provide copy and tamper resistant software [1]. All off-chip memory transactions have to pass through encryption and decryption processes providing a high level of security. Though the original proposal was susceptible to replay attacks, this problem can be addressed with relatively minor changes [25]. However, the main problem with XOM is its significant performance overhead since lengthy encryption and decryption processes reside on the critical path.

Gassend *et al.* [24] propose a hash-tree based verification of untrusted external memory, providing a tamper-proof environment for program execution. The hash-tree based verification dramatically increases memory bandwidth requirements (~logN, where N is the memory size). To address this issue, the authors propose integration of hash-tree mechanism with caches. However, the overall overhead of this approach remains high and its implementation cost may still be prohibitive.

Lu *et al.* [26] propose a similar architecture to that of Gassend, using a tree of message authentication codes (MACs). Cache block MACs are computed from hashes of the cache block, its virtual address, and an application-specific secret key. Higher level nodes of the tree are computed from lower level nodes concatenated with a random number generated from thermal noise. Performance is increased by caching MAC data on the chip. Although this architecture achieves significant performance improvements over that proposed by Gassend *et al.*, it still results in an average performance overhead between 10% and 20%.

Jun Yang *et al.* [3] propose a technique that shifts the cryptographical computation off of the critical memory access path. This scheme uses 'one-time-pad' (OTP) encryption to produce the instruction and data ciphertexts. An I-block is encrypted by XOR-ing it with its encrypted virtual address. Though very efficient, this scheme does not provide software integrity: it fails when an attacker is able to correctly guess instructions in a block, which then can be replaced by a malicious block (the so-called "known plaintext" attack).

Drinic and Kirovski [9] propose a similar approach, but with more cryptographically secure signatures. Each I-block is signed by a keyed MAC that is attached to the I-block. Secure MACs are built using an AES cipher in chain block cipher mode (CBC-MAC). In order to alleviate the effects of signature verification overhead, the authors propose to execute non-critical instructions before they are verified and propose code transformations that will postpone execution of critical instructions. However, these transformations may fail in hiding the crypto latency. In addition, the proposed technique has not been evaluated using a cycle-accurate machine model, so the true potential of the proposed optimizations remain an open issue. Next, the authors do not consider the effects of storing MACs in the I-cache. Finally, this approach does not offer a solution for code confidentiality, nor is it resistant to replay/splicing attacks.

6. Conclusions

This paper presents an efficient, low-complexity hardware technique that provides software integrity and confidentiality. It also presents a scheme for reducing memory overhead, but at the cost of minimal added complexity. The proposed technique improves computer systems' resilience to both software and physical attacks. To achieve low overhead operation of the proposed secure modes, we employ two new approaches: (a) parallelizable instruction block signatures, which are strong double-keyed cryptographic functions providing a high-level of security, yet allowing efficient verification at runtime, and (b) the instruction block verification buffer, which allows instruction execution before the corresponding block is verified, eliminating performance overhead of

runtime verification almost completely. To reduce memory overhead, we utilize the instruction opportunity buffer, which allows instruction blocks larger than the I-cache block size to be protected. The experimental analysis based on a cycle-accurate machine model shows that the proposed solutions impose very low performance and energy overhead.

7. References

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

[2] 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.

[3] J. Yang, L. Gao, and Y. Zhang, "Improving Memory Encryption Performance in Secure Processors," *IEEE Transactions on Computers*, vol. 54, May 2005, pp. 630-640.

[4] M. Milenkovic, A. Milenkovic, and E. Jovanov, "Hardware Support for Code Integrity in Embedded Processors," in *The 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*. San Francisco, CA, USA: ACM Press, 2005.

[5] G. E. Suh, W. O. D. Charles, S. Ishan, and D. Srinivas, "Design and Implementation of the Aegis Single-Chip Secure Processor Using Physical Random Functions," in *Proceedings of the 32nd Annual International Symposium on Computer Architecture*: IEEE Computer Society, 2005.

[6] J. Black and P. Rogaway, "A Block-Cipher Mode of Operation for Parallelizable Message Authentication," in *EUROCRYPT'02*, Amsterdam, Netherlands, 2002, pp. 384-397.

[7] N. Ferguson and B. Schneier, *Practical Cryptography*: John Wiley & Sons, 2003.

[8] T. Zhang, X. Zhuang, S. Pande, and W. Lee, "Anomalous Path Detection with Hardware Support," in *The 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, San Francisco, CA, USA, 2005, pp. 43-54.

[9] 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.

[10] W. Shi, H.-H. S. Lee, M. Ghosh, and C. Lu, "Architectural Support for High Speed Protection of Memory Integrity and Confidentiality in Multiprocessor Systems," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*: IEEE Computer Society, 2004.

[11] W. Shi and H.-H. S. Lee, "Accelerating Memory Decryption with Frequent Value Prediction," in *ACM International Conference on Computing Frontiers*, Ischia, Italy, 2007, pp. 35-46.

[12] "Cadence Aes Cores," <http://www.cadence.com/datasheets/AES_DataSheet.pdf> (Available 2007).

[13] T. Austin, E. Larson, and D. Ernst, "Simplescalar: An Infrastructure for Computer System Modeling," *IEEE Computer*, vol. 35, February, pp. 59-67.

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

[15] 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.

[16] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "Mediabench: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," in *International Symposium on Microarchitecture*, 1997, pp. 330-335.

[17] 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, pp. 27-34.

[18] 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.

[19] 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 2003 2003.

[20] 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.

[21] 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.

[22] 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)*, Portland, OR, USA, 2004, pp. 221-232.

[23] R. B. Lee, P. C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang, "Architecture for Protecting Critical Secrets in Microprocessors," in *International Symposium on Computer Architecture*, Madison, WI, 2005, pp. 2-13.

[24] B. Gassend, G. E. Suh, D. Clarke, M. v. Dijk, and S. Devadas, "Caches and Hash Trees for Efficient Memory Integrity Verification," in *Proceedings of the 9th International Symposium on High-Performance Computer Architecture* Anaheim, California, 2003.

[25] D. Lie, J. Mitchell, C. A. Thekkath, and M. Horowitz, "Specifying and Verifying Hardware for Tamper-Resistant Software," in *IEEE Conference on Security and Privacy*, Berkeley, CA, USA, 2003, pp. 166-177.

[26] C. Lu, T. Zhang, W. Shi, and H.-H. S. Lee, "M-Tree: A High Efficiency Security Architecture for Protecting Integrity and Privacy of Software," *Journal of Parallel and Distributed Computing*, vol. 66, pp. 1116-1128.