Security Extensions for Integrity and Confidentiality in Embedded Processors

Austin Rogers and Aleksandar Milenković
Electrical and Computer Engineering Department
The University of Alabama in Huntsville
301 Sparkman Drive, Huntsville, AL 35899
E-mail: [rogersa, milenka]@ece.uah.edu

Summary

With current trends toward embedded computer systems' ubiquitous accessibility, connectivity, diversification, and proliferation, security becomes a critical issue in embedded computer systems design and operation. Embedded computer systems are subjected to both software and physical attacks aimed at subverting system operation, extracting key secrets, or intellectual property theft. We propose several cost-effective architectural extensions suitable for mid-range to high-end embedded processors. These extensions ensure the integrity and confidentiality of both instructions and data, introducing low performance overhead (1.86% for instructions and 14.9% for data).

Keywords: Secure processors, embedded systems; performance; design.

1 Introduction

Modern society relies on embedded systems to perform an increasing multitude of tasks: they are indispensable to modern communication devices, medical equipment, consumer electronics, home appliances, transportation systems, and even weapons systems. The number of embedded processors far surpasses the number of processors in personal computers and servers, and this gap continues to grow exponentially: 98% of all 32-bit processors sold are used in embedded systems [1]. As the number of embedded applications increases, so do the incentives for attackers to compromise the security of these systems. Security breaches in these systems may have wide ranging impacts, from simple loss of revenue to loss of life. Maintaining security in embedded systems is therefore vital for the consumer, industry, and government.

Depending on the nature of the threat, computer security encompasses three components: confidentiality, integrity, and availability. Confidentiality is violated whenever sensitive or proprietary information is disclosed to any unauthorized entity (human, program, or computer system). Integrity is violated whenever any unauthorized code is executed or unauthorized data is used. Availability is violated whenever an attacker succeeds in denying services to legitimate users. This paper directly addresses the issues of confidentiality and integrity, and indirectly addresses the issue of availability.

Computer systems are often subject to software attacks typically launched across the network by exploiting known software vulnerabilities. According to the United States Computer Emergency Readiness Team [2], 8,064 software vulnerabilities were identified in the year 2006 alone; the number of actual attacks was much greater. These vulnerabilities affect not only personal computers, but also a growing number of portable and mobile computing platforms. Unauthorized copying of software is another major threat. The Business Software Alliance [3]

estimates that, in the year 2006, 35% of all software installed on personal computers was pirated, leading to forty billion dollars in lost revenue. Moreover, embedded systems operating in hostile environments are often subjected to physical attacks. Here adversaries tamper with the memory, buses, and I/O devices in order to extract critical secrets, reverse-engineer the design, or take control of the system. Attackers may also employ side-channel attacks, using indirect analysis to reverse-engineer a system.

Several recent research efforts propose hardware-assisted techniques to prevent execution of unauthorized code [4-8]. These techniques promise higher security, but often fail to counter all attacks, are not suitable for embedded systems or induce prohibitive overheads, or their evaluation does not explore the implications of various implementation choices.

In this paper we propose a cost-effective, flexible architecture for midrange to high-end embedded processors that ensures code and data integrity and confidentiality. An embedded system with a proposed secure processor configured to operate in a secure mode allows execution of trusted programs only (code integrity). These programs accept and process only trusted data (data integrity). Any unauthorized change of either programs or data will be detected. We thus protect against software attacks and physical attacks, such as spoofing, splicing, and replay. In addition, both programs and data can be encrypted, providing privacy (code and data confidentiality).

Integrity is ensured using runtime verification of cryptographically sound signatures embedded in the code and data. Data blocks are further protected from replay attacks by using sequence numbers. The sequence numbers themselves are protected using a tree-like structure. Confidentiality is ensured by encrypting code and data using a variant one-time pad (OTP) scheme. To counter performance overheads induced by signature fetching and verification latencies, the proposed architecture incorporates the following architectural enhancements: parallelizable signatures, conditional execution of unverified instructions, and caching of sequence numbers. Memory overhead due to embedded signatures is reduced by protecting multiple instruction and/or data blocks with a single signature.

The proposed security architecture and corresponding architectural enhancements are modeled with a cycle-accurate processor simulator based on SimpleScalar [9]. The experimental evaluation is conducted by running a set of representative benchmarks while varying relevant architectural parameters. We find that using a combination of architectural enhancements, instruction integrity and confidentiality can be protected with very low performance and power overhead. For example, for a secure embedded processor with 4 KB instruction cache the total performance overhead on a set of benchmarks is 1.86% (ranging from 0% to 3.09%) compared to 43.2% (ranging from 0.17% to 93.8%) for a naïve implementation without any architectural enhancements. The performance overhead induced by protecting data integrity and confidentiality is somewhat higher. For example, a secure embedded processor with 4 KB L1 data cache and 1 KB sequence number cache incurs a total performance overhead of 14.9% (ranging from 0.19% to 41.8%).

Our major contributions are as follows:

- A mechanism for supporting code and data confidentiality with little or no latency by performing the requisite cryptographic operations in parallel with memory accesses.
- A mechanism for reducing code verification overhead by using parallelizable signatures. To our knowledge, we are the first to apply the Parallelizable Message Authentication Code (PMAC) cipher, originally developed by Black and Rogaway [10], to a secure microprocessor architecture.
- A secure and cost-effective mechanism for speculative execution of unverified instructions that protects code integrity and confidentiality with little or no performance overhead.
- A mechanism for protecting data integrity and confidentiality with low overhead.

- A mechanism for reducing memory overhead by protecting multiple instruction and/or data blocks with a single signature.

The remainder of this paper is organized as follows. Section 2 gives a brief overview of threats that computer systems may face. Section 3 describes our proposed architectures for ensuring integrity and confidentiality of both code and data. Section 4 discusses how these architectures are evaluated, and Section 5 presents the results of these evaluations. Section 6 examines related work in the field of hardware-supported security techniques, and Section 7 concludes the paper.

2 Computer Security Threats

This section discusses three classes of attacks to which computer systems may be subjected. We begin with software attacks, and then discuss physical attacks, and side-channel attacks.

The goal of a software attack is to inject malicious code and overwrite a return address so that the injected code is executed. The well-known buffer overflow attack is a classic example of a software attack. The buffer overflow attack takes advantage of insecure code that stores inputs into a buffer without verifying whether or not the buffer's size has been exceeded, allowing the attacker to overwrite data on the stack, including the return address. Other software attacks may exploit integer operation errors [11], format string vulnerabilities, dangling pointers, or cause the program to jump into different sections of code (the so-called arc-injection attack) [12].

Physical attacks involve direct physical tampering. The attacker has access to the address and data buses, and can observe and override bus transactions to perform spoofing, splicing, and replay attacks. In a spoofing attack, the attacker intercepts a bus request and returns a block of his or her choice, which may be malicious. A splicing attack involves the attacker intercepting a bus request and returning a valid but non-requested block. In a replay attack, the attacker returns an old, potentially stale version of the requested block.

Side-channel attacks attempt to gather information about a system via indirect analysis. A side-channel attack consists of two phases: collecting information about the system and analyzing that information to deduce the system's secrets. Some examples of side-channel attacks include timing analysis [13], differential power analysis [14], the exploitation of both intrinsic and induced hardware faults [15], and the exploitation of known architectural features [16].

3 Architectures for Runtime Verification

The proposed runtime verification architectures encompass three stages: secure program installation, secure loading, and secure execution [17]. These stages are illustrated in Figure 1 for the instruction protection architecture. Secure installation modifies an executable binary to work with the verification architecture, producing a secure executable. Secure loading prepares the secure executable to run. The secure execution phase involves the actual execution of the program with runtime verification to ensure integrity and possibly confidentiality. Depending on the desired level of protection, a program may run in an unprotected mode, code integrity only mode (CIOM), code integrity and confidentiality mode (CICM), data integrity only mode (DIOM), data integrity and confidentiality mode (DICM), or some combination of the above.

3.1 Code Integrity and Confidentiality

The secure installation phase of the code protection architecture encompasses key generation, signature generation, and code encryption (if desired). Secure installation must be performed in a special operating mode such as that described by Kirovski *et al.* [5]. The processor must perform secure installations in an atomic manner, and must not reveal any secret information during or after the installation.

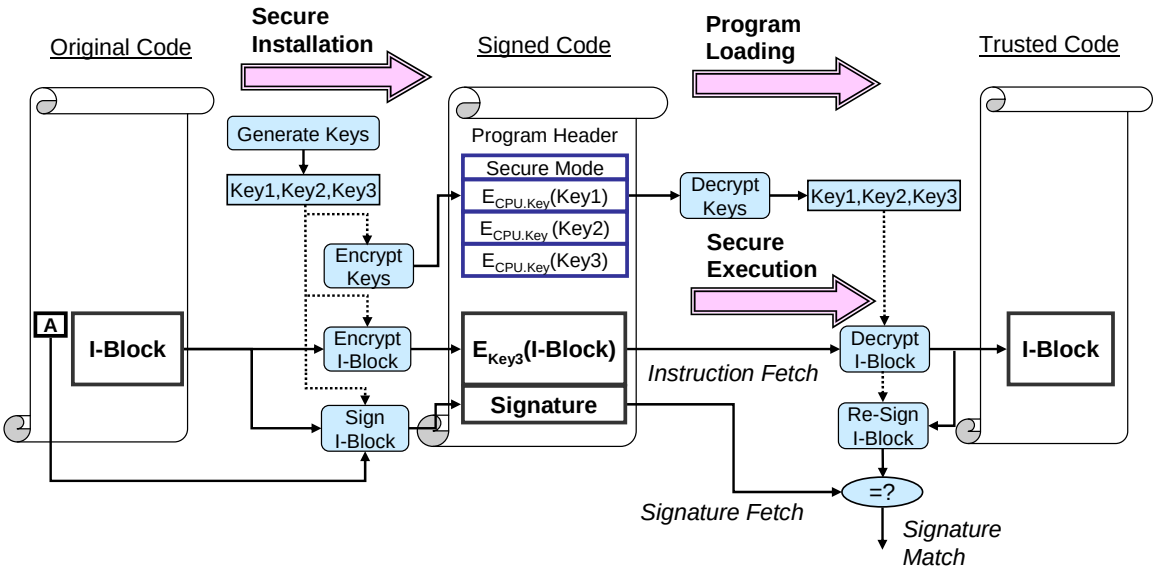


Figure 1. Overview of CIOM/CICM Architecture

Key Generation. Depending on the desired security mode, a program requires zero, two, or three keys (Key1, Key2, Key3). These keys may be generated using thermal noise within the processor chip [18] and/or using a physical unclonable function [19]. These keys are encrypted using a secret key unique to the processor (Key.CPU) and stored in the secure program header. These keys must never leave the processor as plaintext, so key generation and encryption must be performed using only on-chip resources.

Code integrity is ensured by signing each instruction block (I-block). These signatures are calculated during secure installation and embedded in the code of the secure executable, each signature directly after the I-block it protects. An I-block signature is a cryptographic function of the following: (a) the starting virtual address of the I-block or its offset from the beginning of the code section, (b) two of the aforementioned unique program keys, and (c) actual instruction words in the I-block. Using the I-block's address prevents splicing attacks, since I-blocks residing at different addresses will have different signatures, even if the I-blocks themselves are identical. The unique program keys should prevent any execution of unauthorized code, regardless of source. They also prevent the splicing of an instruction block residing at the same virtual address but in another program. Using instruction words is necessary to prevent spoofing and splicing attacks.

The basic implementation of the code protection architecture uses the cipher block chaining message authentication code (CBC-MAC) method [20] to generate signatures. To illustrate the process of signature generation, we assume a 32-bit architecture, 32-byte I-blocks, and 128-bit signatures appended to I-blocks. Each I-block is partitioned into two sub-blocks ($I_{0:3}$) and ($I_{4:7}$). The I-block signature *S* is described in Eq. 1, where *SP* is a secure padding function that pads the 32-bit starting virtual address of the I-block, A_0, to a 128-bit value, and *Key1* and *Key2* are secure program keys. Signatures prevent tampering with the code, but the code can still be inspected by an adversary. To provide code confidentiality, we can expand this scheme with code encryption.

Eq. 1 $$S = AES_{KEY2}[(I_{4:7})\ xor\ AES_{KEY2}((I_{0:3})\ xor\ AES_{KEY1}(SP(A_0)))]$$

Code encryption. Code encryption should provide a high level of security, yet it should not cause significant delays in the critical path during signature verification and code 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* [21]. These three schemes differ in security strength which is still a matter of debate [21, 22]. However, for our implementation, all three schemes have similar hardware complexity and we decided to use the *sign, then encrypt* (StE) scheme.

Our implementation of the StE process is illustrated in Figure 2. In StE, a temporary signature *S* is calculated on plaintext, and then the temporary signature is encrypted, producing the final signature *eS*. The temporary signature is calculated according to Eq. 1. Both instructions and the signature are then encrypted, as described in Eq. 2 and Eq. 3, in which A_i and A_{eS} are the starting virtual addresses of the instruction sub-blocks and signature, respectively. We use Key3 for code encryption because it is recommended that authentication and encryption should not use the same keys [23].

Eq. 2 $(C_{4i:4i+3}) = (I_{4i:4i+3})\ xor\ AES_{KEY3}(SP(A_i)), i = 0..1$

Eq. 3 $eS = S\ xor\ AES_{KEY3}(SP(A_{eS}))$

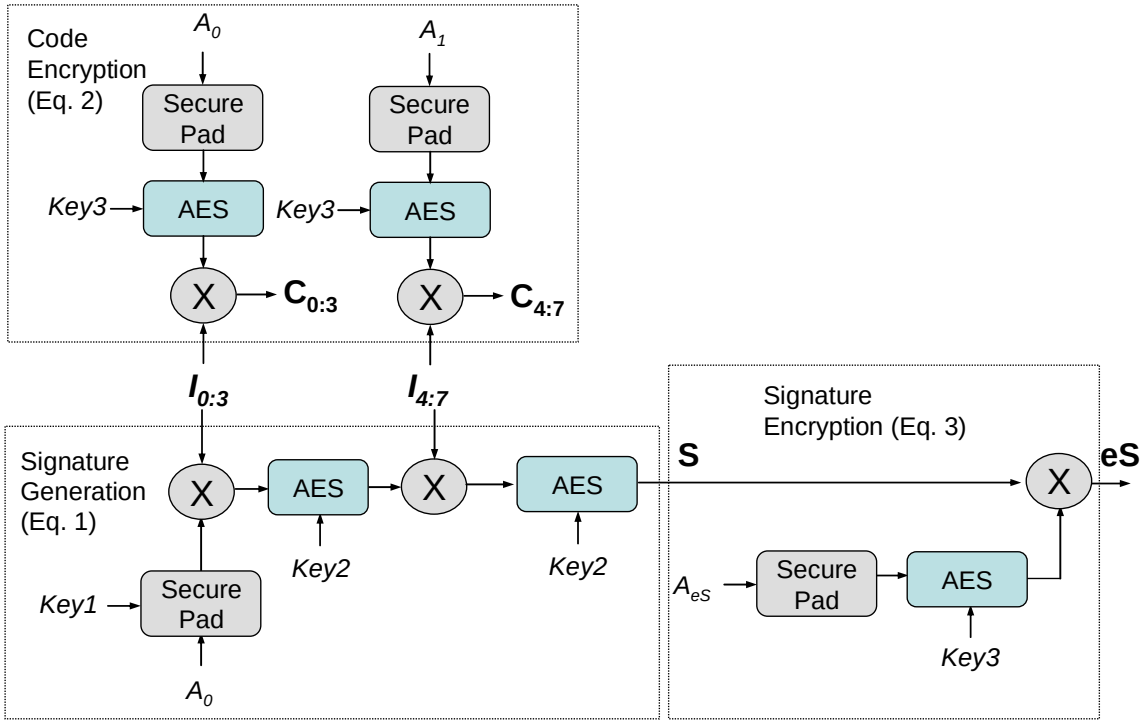


Figure 2. Illustration of Encryption and Signature Calculation using an StE Method

Security considerations. Even with a mechanism that protects code integrity, a skilled attacker can exploit software vulnerabilities to change the target of an indirect jump or return instruction to different existing code sections (so-called arc injection attacks). The CICM 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 [24], 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 nonsigned 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.

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 a cache. Without loss of generality, in the rest of this paper we focus on a system with separate data and instruction first level caches and no second level cache. The instruction cache (I-cache) is a read-only resource, so 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 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. Spoofing and splicing attacks would be detected at this point because they would cause the calculated signature to differ from the fetched signature. Spoofing is detected because the actual code block is included in signature calculation. A successfully spoofed signature would be highly difficult to achieve without knowing the three cryptographic keys. Splicing attacks are detected due to the inclusion of the block address in signature calculation. Cross-executable splicing, where a valid block from one executable is spliced into another executable's address space, would also be detected due to the use of unique program keys for each executable.

Code integrity only mode. Figure 3 shows a timing diagram of operations performed on an I-cache miss. The signature *cS* is calculated using the CBC-MAC cipher [20] as described in Eq. 1. Encryption of the securely padded I-block address (blue boxes in the figure) 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 cryptographic latency is less than the memory access time. Assuming 32-byte I-blocks, 128-bit signatures, 12 clock cycle pipelined AES cipher implementation, 64-bit data bus, and 12/2 memory latency (12 clock for the first chunk, 2 clock cycles for each chunk thereafter), the verification process will be completed in 21 clock cycles, including a cycle for signature comparison. This basic implementation in which the processor stalls until verification is complete is called the Wait-'til-Verified (WtV) scheme.

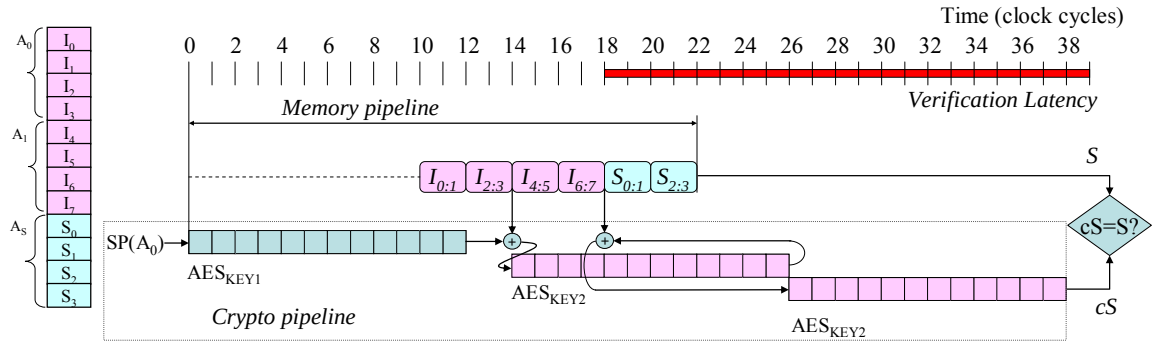


Figure 3. Memory and cryptographic pipeline for runtime verification with the CBC-MAC cipher in CIOM mode.

Code integrity and confidentiality mode. The fetched ciphertext is decrypted according to Eq. 4, producing the original I-block with minimal delay - one XOR operation, since the AES encryption of virtual addresses is overlapped with memory latency. The signature cS is calculated from decrypted instructions. The signature fetched from memory is also decrypted as described in Eq. 5. Three additional cryptographic operations are required on an I-cache miss. As shown in Figure 4, they can be completely overlapped with the memory fetch, thus introducing no additional performance overhead.

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

Eq. 5 $S = eS\ xor\ AES_{KEY3}(SP(A_{eS}))$

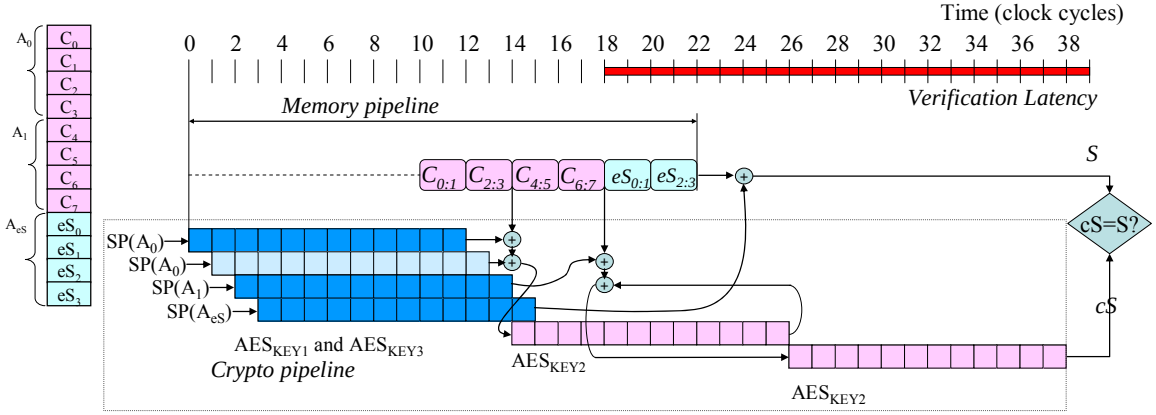


Figure 4. Memory and cryptographic pipeline for runtime verification with the CBC-MAC cipher in CICM mode.

Reducing Performance Overhead. The significant overhead of the basic implementation can be reduced by switching to a Parallelizable Message Authentication Code (PMAC) cipher [10]. Using the PMAC, signatures are calculated on sub-blocks in parallel (Eq. 6), then XOR-ed to create the overall I-block signature (Eq. 7). Encryption and decryption are handled in the same manner as in the basic case.

Eq. 6 $Sig(SB_i) = AES_{KEY2}[(I_{4i:4i+3})\ xor\ AES_{KEY1}(SP(A_i))], i = 0..1$

Eq. 7 $S = Sig(SB_0)\ xor\ Sig(SB_1)$

The runtime verification process using the PMAC cipher in CICM mode is illustrated in Figure 5. As the figure shows, using the PMAC cipher reduces our overhead from 21 to 13 clock cycles per I-cache miss. As with the CBC-MAC implementation, the cryptographic operations required to support decryption do not introduce any additional performance overhead.

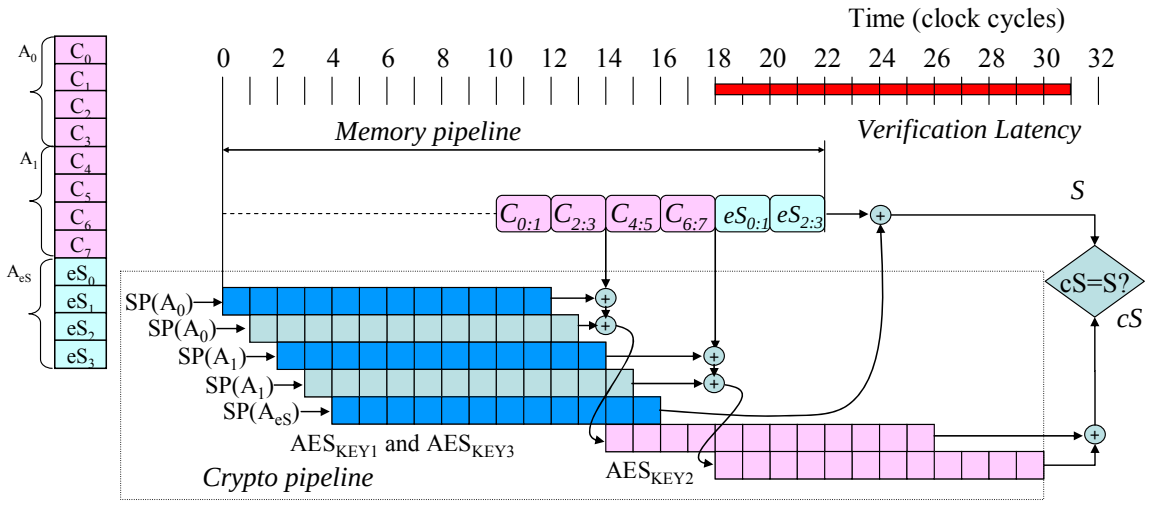


Figure 5. Memory and cryptographic pipeline for runtime verification with the PMAC cipher in CICM mode.

The WtV implementation of the proposed mechanism will not allow execution of instructions before they are verified. 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 mostly hidden 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 6). This buffer is similar to the Sequential Authentication Buffer proposed by Shi *et al.* [25]. 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.

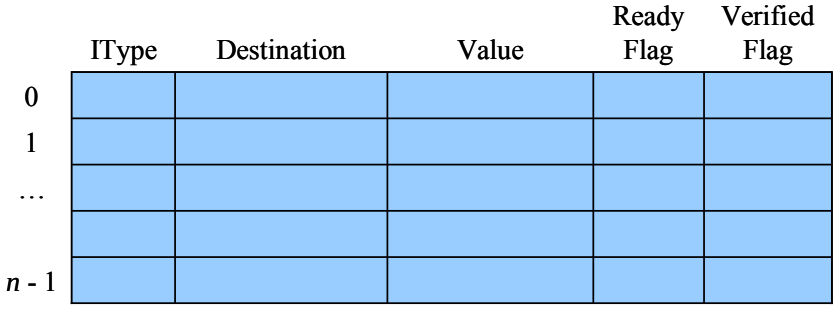
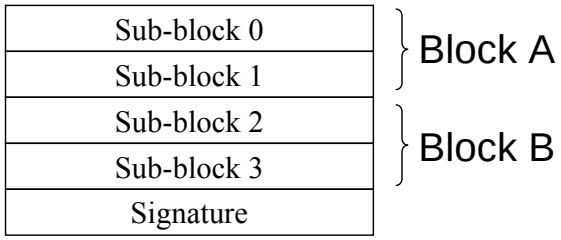


Figure 6. Instruction Verification Buffer

Shi and Lee [26] assert that schemes allowing unverified instructions to execute but not commit until verification is complete may expose sensitive data on the address bus 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. If data confidentiality is desired, then memory access instructions may be stalled until they have been verified.

Reducing Memory Overhead. The proposed architecture could introduce a 50% memory overhead for instructions. In the examples discussed above, for every 32 bytes of instructions, a 16-byte signature is required. This overhead could be prohibitive on embedded systems with tight memory constraints. The solution is to make the protected I-block size a multiple of the I-cache line size. We consider the case where a single signature protects two I-cache blocks, which reduces the instruction memory overhead to 25%.

On an I-cache miss, both I-blocks are required to recalculate the signature. Therefore, the I-cache is probed for the other block. If the other block is not found, then both blocks must be fetched from memory. The layout of these two blocks, which we call Blocks A and B, and their signature is shown on the left side of Figure 7. The right side of the figure enumerates the four possible cases that can be encountered based on which block was missed and whether or not the other block was already available in the cache. After the I-cache miss has been handled, both blocks will then be available in the I-cache.



Miss On	Condition
Block A	Block B not in cache
	Block B in cache
Block B	Block A not in cache
	Block A in cache

Figure 7. Double-block Cases

The first two cases involve a miss on Block A. If Block B is in the I-cache, then the cached version of Block B may be used. Depending on memory latency, however, a continuous fetch of Block A, Block B, and the signature may be faster than starting a new fetch for the signature. The last two cases involve a miss on Block B; if Block A is already in the I-cache then it need not be fetched. Since we use the StE scheme where signatures are calculated on plaintext, we can immediately start signature calculation for blocks already in the I-cache.

3.2 Data Integrity and Confidentiality

The integrity of instructions is protected using signatures crafted to protect against spoofing and splicing attacks. This scheme works well for protecting static data that never change, such as constant data values. Therefore, static data blocks can be protected using the same procedures that protect instructions. Dynamic data that can be programmatically changed are further subject to replay attacks. Therefore, a versioning scheme is required to ensure that all fetched dynamic data is up-to-date.

A tree-like structure is used to defend against replay attacks. This structure is shown in Figure 8. Versioning is implemented on the level of a protected data block. As before, the line size of the lowest level data cache (D-cache) is the most convenient protected block size. Each protected data block will have an associated sequence number. Sequence numbers are stored in a table elsewhere in memory. The sequence number must be included in the formula for the data block signature to protect against replay attacks. Unlike data blocks, sequence numbers need not be encrypted to ensure data confidentiality [6].

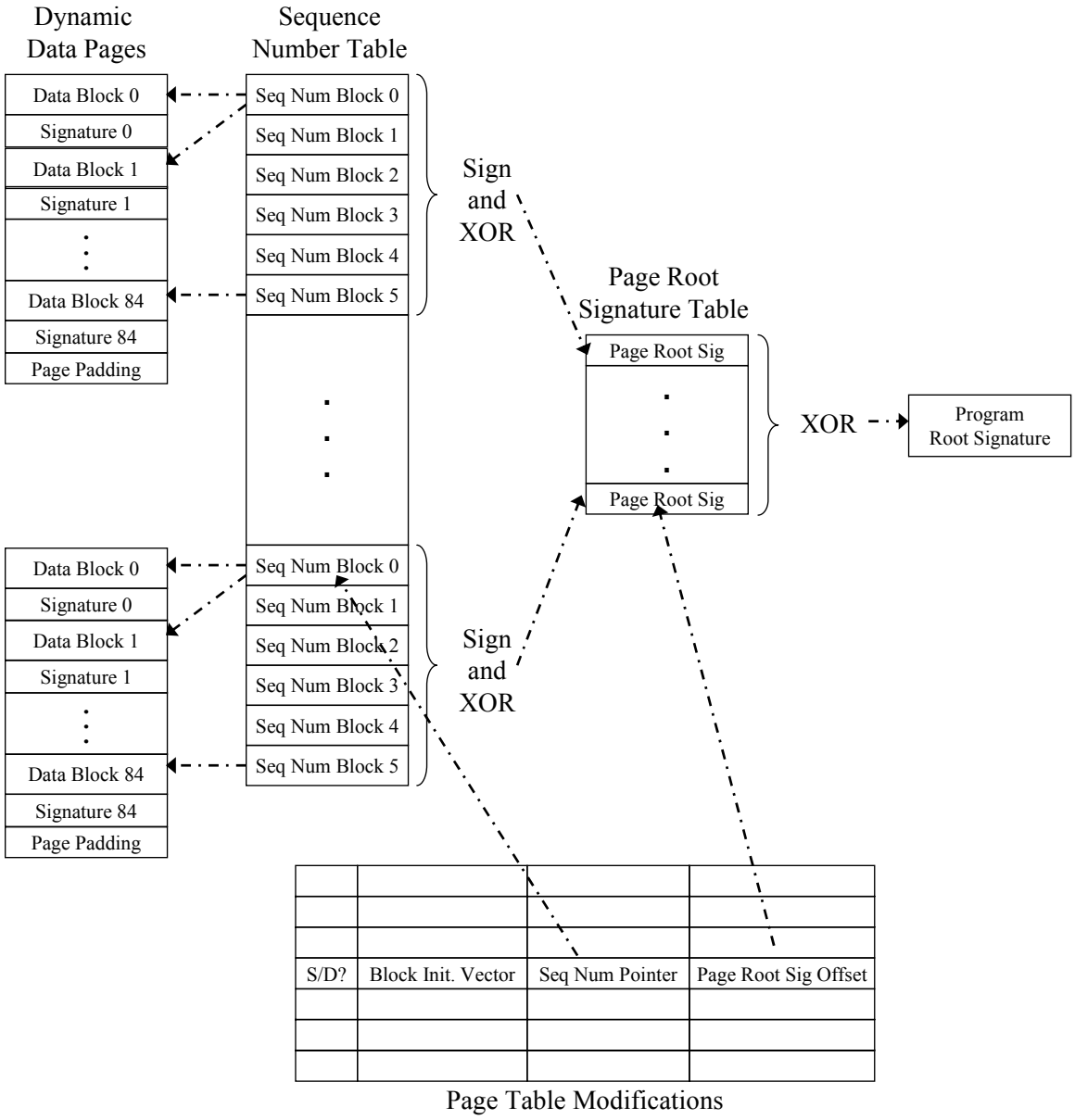


Figure 8. Memory Structures for Protecting Dynamic Data

A sophisticated replay attack could replay sequence numbers as well as data blocks. Therefore, the sequence numbers themselves must be protected against replay attacks. To that end, the sequence number table for a given page is treated as a collection of data blocks, and signatures are calculated for each block. These signatures are then XORed together to form the page root signature. Page root signatures are stored in a separate table in memory.

A final signature is needed to protect the integrity of the page root signatures. This program root signature is calculated by XORing all the page root signatures together. This signature should never leave the processor as plaintext.

We build on the PMAC-WtV scheme to add protection for data. This requires modifications to all three stages of the architectural framework. The bulk of the modifications exist in the secure execution stage, where special handling is required on page allocation, translation lookaside buffer (TLB) misses, D-cache misses, and D-cache evictions. We also propose the use of a special cache for sequence numbers, which requires special handling on a miss.

Secure installation and loading modifications. The secure installation procedure must be modified to sign static data in the same manner as instructions are signed. This is done according to Eq. 6 and Eq. 7. If data confidentiality is desired, static data and their signatures may also be encrypted according to Eq. 3 and Eq. 4. The secure loading procedure must be modified to reset the program root signature in a special register. Since this signature is calculated from the page root signatures of dynamic data pages, it is undefined at load time. On a context switch, the program signature must be re-encrypted using the CPU's private key and stored in the process control block.

Dynamic page allocation. The secure structures required for the data protection architecture must be prepared for each dynamic data page that is allocated. First, its sequence number blocks must be initialized and used to calculate the initial page root signature. The sequence number blocks and the page root signature must be written to memory in their appropriate reserved areas. The starting address or offset from a known starting address for the page's sequence number blocks must be added to the page's entry in the page table (bottom of Figure 8). Secondly, the signatures for the page's data blocks must be calculated and stored in memory.

One way of implementing these procedures is to assume that the operating system is trusted and allow it to perform the necessary operations on memory allocation. This approach could potentially introduce high overhead. The other option is to perform the operations in hardware and allow the OS to trigger them using a special instruction. We choose the latter option for both procedures.

We assume a page size of 4 KB for our example architecture. Each page contains 85 data blocks with their 16-byte signatures, with 16 bytes of padding required at the end of the page. A 2-byte sequence number is assigned to each data block. Thus, a total of six 32-byte blocks are required for sequence numbers protecting a dynamic page. These blocks are stored in a reserved location in memory called the sequence number table (Figure 8).

The page root signature for a new dynamic page must be calculated from the page's sequence number blocks. Each sequence number block is divided into two sub-blocks, $SQ_{0:3}$ and $SQ_{4:7}$, and their signatures are calculated according to Eq. 8. Only one sub-block of the sixth sequence number block is used; the other sub-block may be neglected in signature calculations. The signatures of each sequence number sub-block are XORed together to form the page root signature. Once calculated, the page root signature is stored in the page root signature table. The index of the page root signature in the table is stored in the page table (see the bottom of Figure 8).

Eq. 8 $$Sig(SB_i) = AES_{KEY2}[(SQ_{4i:4i+3})\ xor\ AES_{KEY1}(SP(A_i))], i = 0..1$$

The program root signature is calculated by XORing the page root signatures of all dynamic data pages. Thus, when a new dynamic data page is allocated, the program root signature must be updated by XORing it with the newly calculated page root signature. All calculations on the program root signature must be performed on-chip.

The other task required for new dynamic data pages is data block signature initialization. This could be done on page allocation at the cost of significant overhead. Instead, we create the signatures on the block's first write-back. A block initialization bit vector must be established with a bit for each data block in the new page, specifying which data blocks in the page have been used. Each block is initially marked as unused. The block initialization bit vector is stored in the page table.

TLB miss and write-back. On a TLB miss, information about a data page is brought into the TLB. If the page in question is a dynamic data page, the extra security data required by this architecture must be loaded from the page table and stored in the TLB at this point: a bit specifying whether this page is static or dynamic, the starting address (or offset from a known starting address) of the page's sequence number blocks, the index of the page root signature associated with this page, and the page's block initialization bit vector. The integrity of the page

root signatures is also verified at this point. The page root signatures from every active dynamic data page are retrieved from the TLB or from memory. These signatures are XORed together to recalculate the current program root signature. If the calculated program root signature does not match the one stored on-chip, then the page root signatures have been subjected to tampering and a trap to the operating system is asserted. The upper bound of the performance overhead introduced on a TLB miss is the time required to fetch all page root signatures from memory.

A page root signature will be updated when the sequence number for a data block within that page is incremented. The program root signature will also be updated at that time. TLB write-backs add negligible overhead. If the page root signature contained in the entry to be evicted is not dirty, then no operations are required. If it is dirty, the only required operation is to place the appropriate page root signature and bit initialization vector into the write buffer.

Data Cache Miss. Data block verification is performed on data cache read misses and write misses on blocks that have already been used. Therefore, on a write miss the first task is to check the block's entry in the block initialization bit vector in the TLB. If the block has not yet been used then no memory access is required. The cache block is simply loaded with all zeros, preventing malicious data from being injected at this point.

If the miss is a read miss or a write miss on a previously used block, then the data block must be fetched and verified. The signatures of the sub-blocks $D_{0:3}$ and $D_{4:7}$ fetched from memory are calculated in the same manner as static data sub-blocks and instruction sub-blocks. If the block is in a dynamic page, the sequence number SN^j must be fetched and encrypted before the signature cS of the entire block may be calculated (Eq. 9 and Eq. 10). Therefore, fetching the sequence number is in the critical path of data verification. The inclusion of sequence numbers in signature calculation ensures that stale signatures will not match the current signature, causing replay attacks to be detected at this point. The handling of sequence numbers is discussed below. The simplest implementation stalls the processor until data block verification is complete. We assume this simple implementation throughout the rest of the paper, as speculatively using the data would require more complex hardware than the one used for speculative execution of unverified instructions.

Eq. 9 $SN^{j*} = AES_{KEY1}[SP(SN^j)]$

Eq. 10 $cS = Sig(SB_0)\ xor\ Sig(SB_1)\ xor\ SN^{j*}$

If the architecture is running in DICM mode, then the fetched data block and signature must be decrypted. Decryption is performed according to Eq. 4 and Eq. 5. However, in the case of dynamic data, the secure padding function SP must include the sequence number. The resulting pad must be unique to prevent pad reuse.

The first task that must be performed on a data cache miss is to request the appropriate sequence number from memory. Once the sequence number is available, the verification latency is the same as the PMAC-WtV case discussed above. The verification timing is shown in Figure 9. This figure shows DIOM mode for brevity; DICM would not introduce any additional latency as it only requires two additional cryptographic operations prior to starting signature generation. This would shift the start of signature generation for the first block by one clock cycle, but would not affect the overall latency. As the figure shows, signature verification is complete after 31 clock cycles (measured from the time at which the sequence number is available), at which time the processor may continue and use the fetched data.

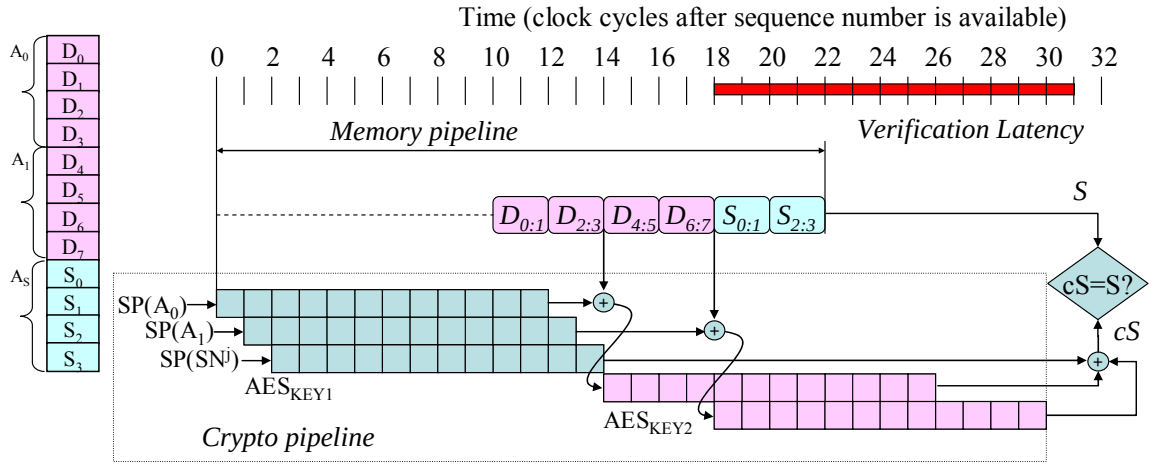


Figure 9. Memory and cryptographic pipeline for runtime verification of dynamic data block with the PMAC cipher in DIOM mode.

Data cache write-back. When a dirty data block from a dynamic data page is chosen for eviction, the signatures of its sub-blocks are calculated according to Eq. 6. The sequence number must also be updated. The current sequence number SN^{j} must be fetched and incremented according to Eq. 11. The new sequence number $SN^{(j+1)}$ is then encrypted as described in Eq. 12, and used to calculate the new signature for the total data block as in Eq. 13. Again, the sequence number is on the critical path for signature generation, and must be handled appropriately. At this point, the page root signature must also be updated. The signature of the appropriate sequence number sub-block must be calculated prior to the sequence number increment. This signature is then XORed with the page root signature contained in the TLB, effectively subtracting it out of the signature. The signature of the sequence number sub-block after the increment is also calculated and XORed with the page root signature, which is stored back in the TLB. A similar procedure is followed to update the program root signature using the old and new page root signatures.

Eq. 11 $SN^{(j+1)} = SN^{j} + 1$

Eq. 12 $SN^{(j+1)^*} = AES_{KEY1}[SP(SN^{(j+1)})]$

Eq. 13 $S = Sig(SB_0)\ xor\ Sig(SB_1)\ xor\ SN^{(j+1)^*}$

When running in DIOM mode, D-cache write-back requires eight cryptographic operations, for a performance overhead of 19 clock cycles. DICM mode requires three additional operations to encrypt the data block and its signature (see Eq. 3 and Eq. 4), raising the overhead to 22 clock cycles. As mentioned above, the secure padding function must include the block's sequence number to prevent pad reuse.

Pad reuse is also a concern in the case of a sequence number overflow. The preceding discussion assumed an application where write-back frequencies are low enough so that no sequence number will overflow. If this is not the case, then the sequence number size must be increased to make overflows highly improbable. A split sequence number scheme such as that proposed by Yan et al. [27] may be implemented. In this scheme, one large major sequence number is associated with several smaller minor sequence numbers. Each block's individual sequence number consists of a concatenation of one minor sequence number and its associated major number. When a minor sequence number overflows, its major number must be incremented and all data blocks protected by that major number's other minor numbers must be re-signed and re-encrypted.

Sequence number cache miss and write-back. Since sequence numbers are on the critical path for both data cache misses and write-backs, efficient handling of sequence numbers is imperative to keep performance overhead low. Thus we cache sequence numbers on-chip as in [6], preventing extra memory accesses on each data cache miss or write-back.

On a sequence number cache miss, the six sequence number blocks associated with the page that caused the miss must be retrieved. Some of these may be already cached; the rest must be fetched from memory. The implementation must balance overhead versus complexity. For our sample implementation, we choose a scheme of moderate complexity. On a sequence number cache miss, the sequence number cache is probed for the page's first sequence number block. If it is found in the cache, the cache is probed for the next block and so forth until a block is not found in the cache. A memory fetch is initiated for that block, and further probing for the rest of the blocks occurs in parallel with the memory operation. All blocks between and including the first not found in the cache to the last not found in the cache are fetched from memory. Any fetched blocks that were found in the sequence number cache are ignored, and the blocks that were not previously cached are inserted in the cache. The overhead incurred on a sequence number cache miss is thus the time required to fetch the necessary sequence number blocks plus the time required for one cryptographic operation (12 clock cycles in our example system) to calculate the signature on the final sequence number sub-block that is fetched.

The signatures for each sub-block of the sequence number blocks are calculated according to Eq. 8, and then XORed together to calculate the page root signature. This recalculated page root signature is checked against that stored in the TLB. If they do not match, then a trap to the operating system is asserted.

When sequence number blocks are evicted from the sequence number cache, no cryptographic activity is required. Furthermore, the page root signature is updated during data cache write-back, and will be written to memory during a TLB write-back. Sequence number cache write-backs introduce negligible overhead. As with TLB write-backs, no cryptographic operations are required. The sequence number block being evicted only needs to be placed in the write buffer to be written to memory when the bus is available.

Reducing memory overhead. In our example architecture, we use 32-byte protected blocks with 16-byte signatures. This leads to a memory overhead of 50% due to signatures. As with the instruction protection architecture, we reduce this overhead by protecting two D-cache blocks with a single signature. This increases the number of D-cache blocks in each 4 KB data page from 85 to 102 (52 protected blocks). Since sequence numbers are associated with protected blocks rather than with cache lines, the number of 32-byte sequence number blocks required per page is reduced to four. This also reduces the upper bound on the overhead incurred on sequence number cache misses.

D-cache read misses with double-sized protected blocks are subject to the same four cases described in Figure 7. They are thus handled similarly to I-cache misses with double-sized protected blocks, with the additional requirement of the protected block's sequence number for signature calculation. D-cache write-backs also require both blocks to calculate the new signature. If the other block is not in the cache, it may be fetched in parallel with the required cryptographic operations described above. In our sample architecture, four additional cryptographic operations are required to calculate the signature of the other block, the last of which may not begin until the other block is fully available.

4 Experimental Environment

Simulator. The simulator used to evaluate the performance and energy of the proposed architectures is a derivative of the Sim-Panalyzer ARM simulator [28]. Sim-Panalyzer is itself an extension of sim-outorder, the most detailed simulator from the SimpleScalar suite [9]. The simulator performs a full functional simulation providing a cycle-accurate timing analysis for both

the instruction and data protection architectures and an estimate of the energy overhead for the instruction protection architecture.

As a measure of performance we use a normalized execution time calculated as the number of clock cycles a benchmark takes to execute on a processor with security extensions divided by the number of clock cycles the benchmark takes to execute on the baseline configuration. The energy overhead is determined by dividing the total energy spent by the processor with security extensions by the total energy spend by the baseline configuration.

Workload. As a workload for performance and energy analysis we use a set of benchmarks for embedded systems taken from the MiBench [29], MediaBench [30], and Basicrypt [31] benchmark suites. Since signature verification is done only at cache misses, the benchmarks selected from these suites have a relatively high number of cache misses for at least one of the simulated cache sizes. Thus, these benchmarks often represent a worst-case scenario with the greatest possible overhead. Table 1 lists the selected benchmarks, the total number of executed instructions, and the number of I- and D-cache misses per 1000 executed instructions when executed on the baseline architecture.

Table 1. Cache Miss Rates for Embedded Benchmarks

Benchmark	IC [10^6]	Instruction Cache Misses per 1000 Executed Instructions				Data Cache Misses per 1000 Executed Instructions			
		1 KB	2 KB	4 KB	8 KB	1 KB	2 KB	4 KB	8 KB
blowfish_enc	544.0	33.8	5.1	0	0	63.5	43.4	8.4	0.3
cjpeg	104.6	7.6	1.3	0.3	0.1	92.5	69.8	56.9	8.9
djpeg	23.4	11.9	5.5	1.3	0.3	88	54.3	34.8	13.4
ecdhb	122.5	28.5	8.5	2.9	0.1	5.7	1.2	0.3	0.2
ecelgencb	180.2	25.4	4.5	1.4	0.1	3	0.7	0.2	0.1
ispell	817.7	72.4	53	18.8	2.9	60.4	33.4	4.3	1.5
mpeg2_enc	127.5	2.2	1.1	0.4	0.2	54.6	30.2	6.7	1.7
rijndael_enc	307.9	110.2	108.3	69.5	10.3	227.5	190.9	111.5	15.2
stringsearch	3.7	57.7	35	6.2	2.4	87.6	43	7.3	4.3

Simulator parameters. The simulator is configured to simulate an ARM architecture running at 200 MHz. The I/O supply voltage is 3.3 V, with an internal logic power supply of 1 V. All other power-related parameters correspond with a 0.18 μm process, and are obtained from a template file provided with Sim-Panalyzer. All simulated systems are assumed to have separate Level 1 instruction and data caches of the same size. This size varies between 1 KB, 2 KB, 4 KB and 8 KB. All cache line sizes are taken to be 32 bytes, as every benchmark exhibited better performance on a baseline system with 32-byte cache lines than with 64-byte lines. All caches use the least recently used (LRU) replacement policy. For RbV implementations, instruction verification buffer depth is 16 unless otherwise noted. Other architectural parameters used in the simulations are described in Table 2. The energy consumed by the pipelined cryptographic hardware is modeled as that caused by 57,000 gates of combinational logic [32].

Table 2. Simulation Parameters

Simulator Parameter	Value
Branch predictor type	Bimodal
Branch predictor table size	128 entries, direct-mapped
Return address stack size	8 entries
Instruction decode bandwidth	1 instruction/cycle
Instruction issue bandwidth	1 instruction/cycle
Instruction commit bandwidth	1 instruction/cycle
Pipeline with in-order issue	True
I-cache/D-cache	4-way, first level only
I-TLB/D-TLB	32 entries, fully associative
Execution units	1 floating point, 1 integer
Memory fetch latency (first/other chunks)	12/2 cycles and 24/2 cycles
Branch misprediction latency	2 cycles
TLB latency	30 cycles
AES latency	12 clock cycles
Address translation (due to signatures)	1 clock cycle
Signature comparison	1 clock cycle

5 Results

This section presents analysis results for the proposed architectures. We start with a discussion of the on-chip complexity of the architectures and memory overhead, and then analyze the simulation results for performance and energy overhead.

5.1 Complexity

The proposed security extensions require state machines for performing various tasks, logic for address translation, the instruction verification buffer, various other buffers and registers, hardware for key generation, and a pipelined cryptographic unit. All but the last two require relatively little additional on-chip area. A physical unclonable function (PUF) unit for key generation requires nearly 3,000 gates [19]. The pipelined cryptographic unit, following the commercially available Cadence high performance 128-bit AES core [32], requires approximately 57,000 logic gates. An additional source of complexity is the sequence number cache; its complexity is determined by its size and organization, which are design parameters.

5.2 Memory

The memory overhead incurred by the instruction protection architecture is a simple function of the protected block size and the number of instruction blocks in the program. Each signature is 16 bytes long. If 32-byte protected blocks are chosen, then the size of the executable segment of the program increases by 50%. This overhead is reduced to 25% for 64-byte protected blocks, and to 12.5% for 128-byte protected blocks. Depending on page size, page padding may also be required. For instance, with a 4 KB page size, 32-byte protected blocks, and 16-byte signatures, 16 bytes of padding are required for each page.

The data protection architecture incurs overhead at different rates for pages containing static data and pages containing dynamic data. The memory overhead for protecting static data blocks follows the figures given for instruction blocks. For dynamic data the overhead is slightly larger due to additional space needed in the page table and the sequence number table. The size of the sequence number blocks is a design parameter; our sample architecture requires 6 sequence number blocks of 32 bytes each, for a total sequence number overhead of 192 bytes per protected dynamic page.

5.3 Performance

CICM. The normalized execution times of the embedded benchmarks running in CICM mode are plotted in Figure 10, and expressed numerically in Table 3. We analyze the following implementations: CBC-MAC WtV, PMAC WtV, and PMAC RbV with each signature protecting a single I-block. We also show results for PMAC RbV with each signature protecting two I-blocks ("Double" case in the figure and tables) and caching all fetched I-blocks. These plots clearly show that the PMAC RbV implementation incurs the lowest performance overhead (negligible in most cases). For example, the total performance overhead across all benchmarks for PMAC-RbV is only 1.86%, ranging from 0% to 3.09%, compared to 43.2%, ranging from 0.17% to 93.8%, with CBC-MAC. PMAC-RbV performs very well even with very small instruction caches; for example with 1 KB instruction cache, the total overhead is 3.70%, compared to 91.2% observed for CBC-MAC. The results also indicate that two I-cache blocks can be protected by a single signature without incurring further overhead. In fact, some benchmarks exhibit a speedup relative to baseline performance due to the prefetching behaviour of caching all fetched I-blocks.

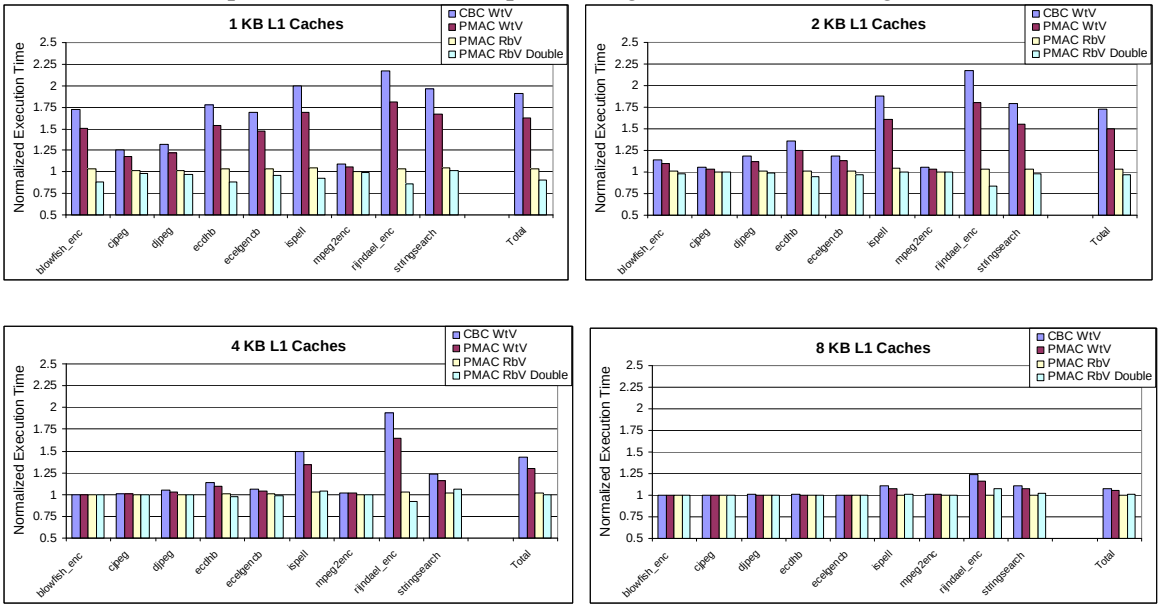


Figure 10. CICM Performance Overhead

Table 3. CICM Performance Overhead

	Performance Overhead [%]															
	CBC WtV				PMAC WtV				PMAC RBV				PMAC RBV Double			
Size [KB]	*1*	*2*	*4*	*8*	*1*	*2*	*4*	*8*	*1*	*2*	*4*	*8*	*1*	*2*	*4*	*8*
blowfish_enc	72.6	13.7	0.17	0.00	50.2	9.46	0.14	0.00	3.14	0.60	0.09	0.00	-12.3	-1.70	0.06	-0.01
cjpeg	25.3	5.23	1.13	0.29	17.5	3.60	0.77	0.19	1.14	0.23	0.00	0.00	-1.52	0.35	0.06	-0.00
djpeg	31.9	18.0	4.89	0.88	21.9	12.3	3.31	0.53	1.23	0.64	0.07	0.00	-2.51	-1.41	-0.21	-0.09
ecdhb	77.5	35.7	14.2	0.78	53.6	24.7	9.85	0.54	3.49	1.46	0.64	0.03	-12.0	-5.94	-1.96	-0.06
ecelgencb	68.8	18.8	6.58	0.34	47.6	13.0	4.55	0.24	3.14	0.77	0.30	0.01	-4.12	-3.10	-0.91	-0.04
ispell	99.8	88.5	49.7	10.4	68.8	60.9	34.3	7.22	4.31	4.03	2.60	0.50	-7.68	0.48	4.29	1.33
mpeg2enc	8.70	5.05	2.16	1.09	6.00	3.50	1.50	0.76	0.39	0.24	0.13	0.09	-0.58	-0.39	-0.05	-0.02
rijndael_enc	117.4	117.7	93.8	24.1	80.6	80.9	64.5	16.2	3.62	3.77	3.09	0.00	-14.2	-16.1	-8.34	7.53
stringsearch	96.6	79.3	23.7	10.4	66.7	54.9	16.4	7.16	4.33	3.80	1.36	0.30	1.79	-2.03	5.75	1.75
Total	**91.2**	**72.7**	**43.2**	**7.65**	**62.8**	**50.1**	**29.8**	**5.23**	**3.70**	**2.97**	**1.86**	**0.16**	**-9.37**	**-3.72**	**-0.18**	**1.52**

The CICM architectures with single-sized protected I-blocks introduce a fixed amount of verification latency on each I-cache miss. Assuming that this latency dominates other contributions to performance overhead, a linear relationship between the performance overhead and the I-cache miss rate is expected. Figure 11 plots the normalized execution time versus baseline I-cache miss rate for the CBC WtV, PMAC WtV, and PMAC RbV single-sized protected block CICM implementations. The plot includes data points for all cache sizes. As expected, CBC-WtV and PMAC-WtV plots exhibit some linearity and the overhead is directly proportional to the number of I-cache misses. However, PMAC-RbV implementation shows excellent results even with a significant number of cache misses. This plot also helps us quantitatively evaluate the contribution of each enhancement individually (a) replacing CBC-MAC with PMAC and (b) allowing speculative execution of unverified instructions, across the design space.

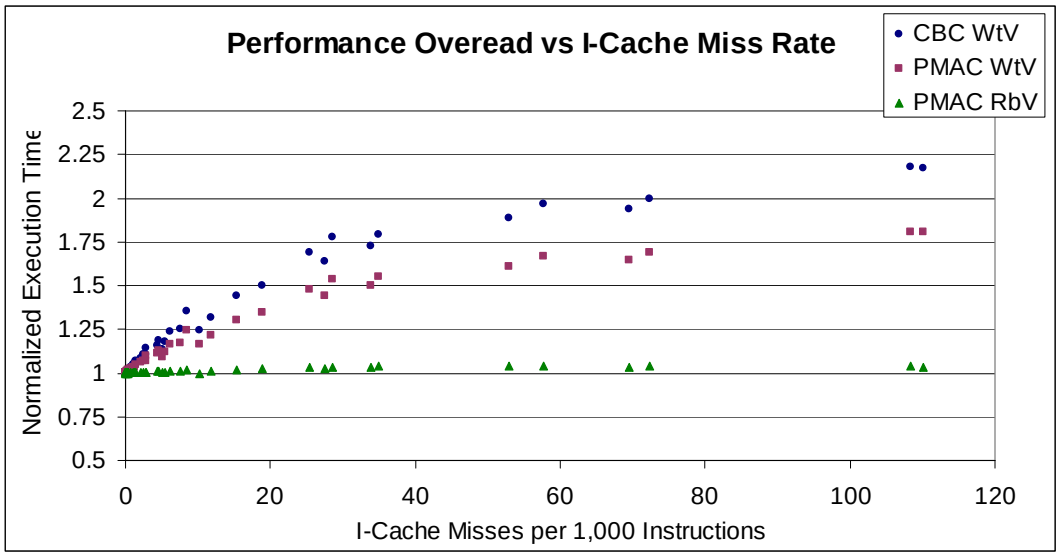


Figure 11. CICM Performance Overhead vs. Baseline Instruction Cache Miss Rate

We choose two benchmarks with relatively high I-cache miss rates, *rijndael_enc* and *ispell*, for exploring the optimum IVB depth. They are simulated in the CICM mode using the PMAC cipher with each signature protecting a single I-block. IVB depth is varied from two to 32 entries in powers of two. The normalized performance overheads from these experiments are plotted in Figure 12. For both benchmarks, the greatest performance improvement is observed when the IVB depth is increased from 8 to 16. Further increasing the IVB depth yields minimal improvement. Thus, a 16-entry IVB, capable of holding 2 cache lines, appears to be optimal. This result can be explained as follows. A potential worst-case scenario for the IVB would be an I-cache miss, followed by the linear execution of the new cache line's instructions, and then followed by the linear execution of another line already in the cache. This worst case scenario results in 16 instructions being placed in the IVB. Even with an ideal processor efficiency of 1 cycle per instruction, the first 8 instructions would be verified before execution of the second 8 instructions is completed.

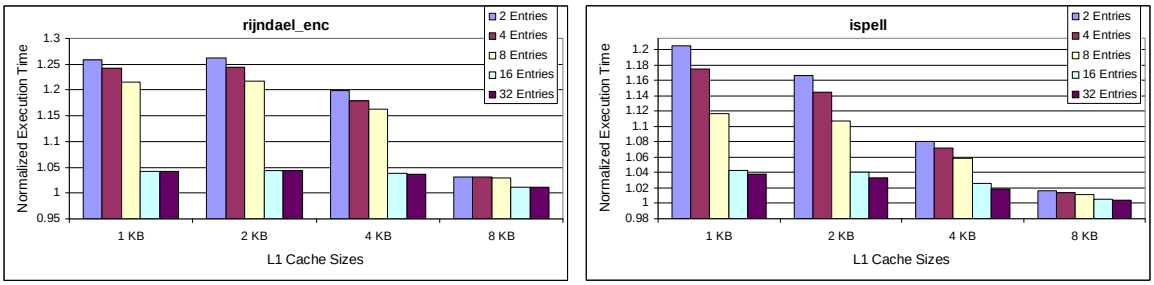


Figure 12. Optimal Instruction Verification Buffer Sizing

DICM. Performance overhead is also evaluated for the data protection architecture. We use the most efficient CICM implementation, PMAC RbV, which has been shown to introduce negligible performance overhead. The normalized execution times for embedded benchmarks running in CICM/DICM mode (protecting both integrity and confidentiality of code and data) are shown in Figure 13 and presented numerically in Table 4. Results are shown for the various cache sizes of interest, and for both single protected D-cache block and double protected D-cache block cases. For each cache size, the sequence number cache size is varied between 25% and 50% of the data cache size. All sequence number caches are 4-way set associative.

The results show that the DICM architecture incurs significant overhead for small D-cache sizes. This overhead greatly decreases as D-cache size increases; all benchmarks exhibit less than 25% performance overhead with an 8 KB D-cache. They also indicate that larger sequence number caches significantly reduce the performance overhead of most benchmarks on systems with small D-caches, but offer little improvement for systems with larger D-caches. Thus the choice of sequence number cache size should be driven by the expected workload of the system and the overall cache budget. Systems with larger D-caches and smaller sequence number caches tend to outperform systems with smaller D-caches and larger sequence number caches. For instance, a simulated hypothetical system with a 5 KB D-cache and 1 KB sequence number cache exhibits total performance overhead of 8.86% compared to a base system with a 4 KB D-cache, whereas a system with 4 KB D-cache and 2 KB sequence number cache exhibits 11.0% overhead.

Given the potentially significant performance overhead from protecting data, the DICM architecture is obviously not suitable for embedded systems that are under severe performance constraints. Additionally, not all applications require full protection of dynamic data, and may tolerate only code protection. A partial DICM implementation may be achieved by not using sequence numbers and generating cryptographic pads and signatures for dynamic data in the same manner as for code. This would not protect against replay attacks, and a dedicated attacker could

compromise integrity by taking advantage of the resulting pad reuse. However, the performance overhead of such a scheme would be on par with that resulting from only protecting code.

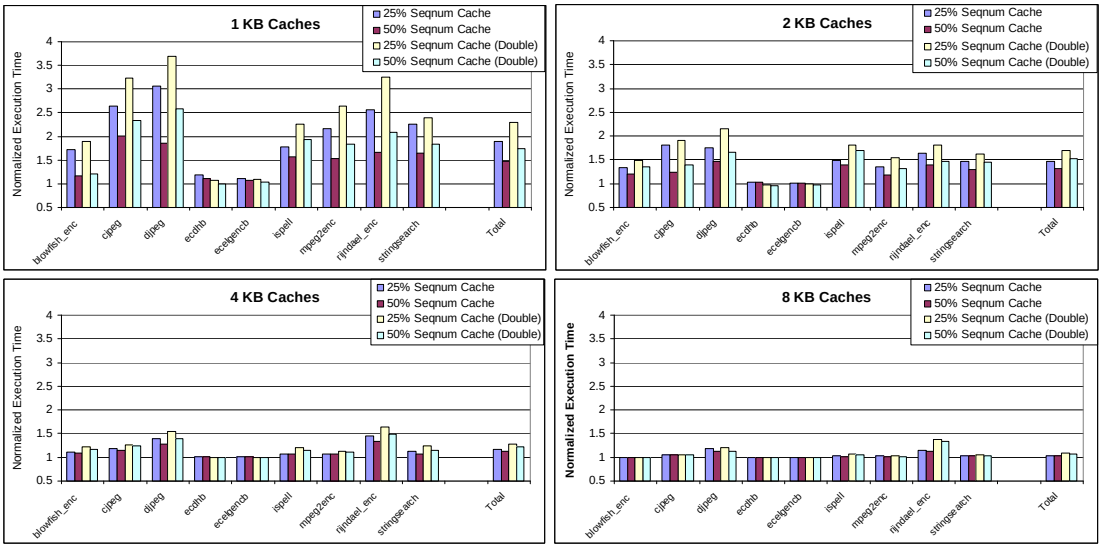


Figure 13. CICM/DICM Performance Overhead

Table 4. CICM/DICM Performance Overhead

	Performance Overhead [%]															
	Sequence Number Cache Size 25% of L1 Data Cache Size								Sequence Number Cache Size 50% of L1 Data Cache Size							
	Single Protected Block				*Double Protected Block*				*Single Protected Block*				*Double Protected Block*			
Size [KB]	*1*	*2*	*4*	*8*	*1*	*2*	*4*	*8*	*1*	*2*	*4*	*8*	*1*	*2*	*4*	*8*
blowfish_enc	69.5	33.5	10.4	0.02	89.8	49.6	21.7	0.03	13.3	20.0	9.14	0.02	20.4	35.8	16.9	0.02
cjpeg	162.9	81.0	18.0	5.09	223.3	91.2	27.0	5.71	100.4	24.4	15.3	4.52	133.7	38.8	24.1	4.94
djpeg	204.3	75.7	40.2	18.0	269.3	115.1	55.1	20.4	85.3	45.8	28.7	13.0	158.6	66.1	39.0	13.0
ecdhb	15.0	2.60	0.41	0.26	7.07	-1.50	-1.22	0.27	7.92	1.15	0.26	0.24	-0.74	-3.75	-1.43	0.24
ecelgencb	7.11	1.30	0.19	0.05	9.26	-0.52	-0.59	0.03	3.47	0.66	0.15	0.03	2.7	-1.89	-0.67	0.01
ispell	73.6	43.9	5.28	2.58	126.3	81.1	20.1	7.33	52.4	34.9	3.61	1.60	93.0	69.9	12.0	5.04
mpeg2enc	115.9	34.9	7.88	2.55	163.3	54.6	12.6	2.84	53.3	19.0	6.27	2.02	84.8	31.8	9.94	2.21
rijndael_enc	153.2	59.7	41.8	15.3	225.4	82.1	63.9	36.8	63.9	34.9	29.9	13.8	108.8	46.9	48.9	34.0
stringsearch	121.9	43.2	10.7	3.22	139.5	62.9	24.1	4.33	60.2	26.4	4.85	2.92	84.0	46.0	13.9	3.91
Total	**85.2**	**43.3**	**14.9**	**3.49**	**130.3**	**70.0**	**28.9**	**8.31**	**44.0**	**29.4**	**11.0**	**2.80**	**74.9**	**52.6**	**22.1**	**6.86**

5.4 Energy

CICM. The normalized values of energy consumed by the microarchitecture running in CICM mode are plotted in Figure 14 and shown numerically in Table 5. Results are presented for cache sizes of 1 KB, 2 KB, 4 KB, and 8 KB, and for the following implementations: CBC-MAC WtV, PMAC WtV, PMAC RbV, and PMAC RbV with double-sized protected blocks and caching all fetched I-blocks. The plots follow the normalized execution time plots very closely, showing a

strong correlation between execution time and energy consumed. Once again, PMAC RbV is the most efficient of the single-sized protected block implementations, and the double-sized protected block implementation introduces little or no additional overhead. As before, some benchmarks benefit from the prefetching behavior in the double-sized protected block implementation, consuming less energy due to shorter runtimes.

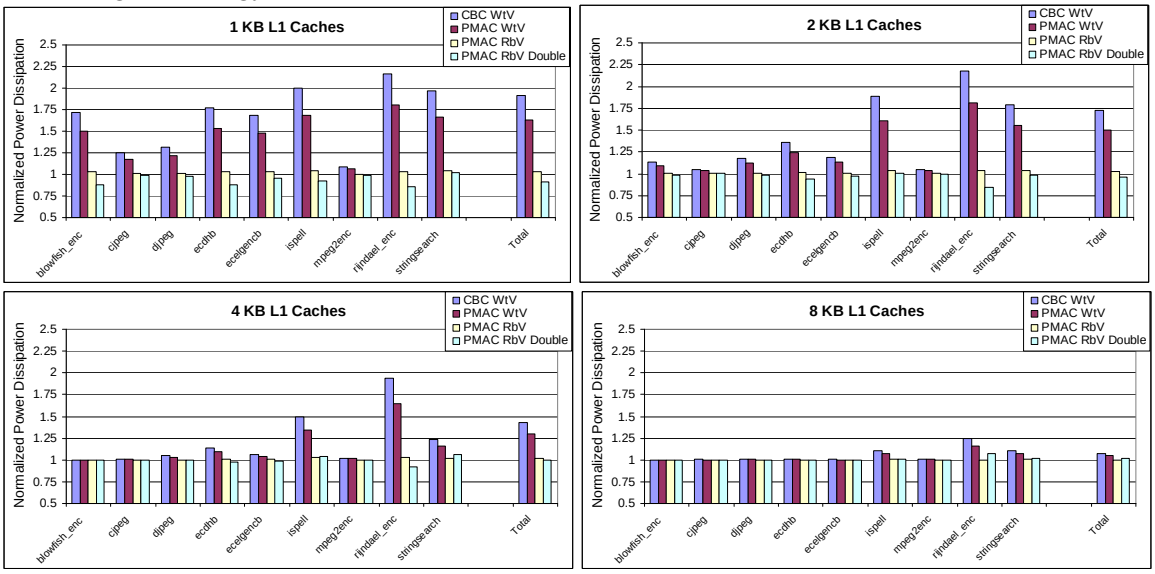


Figure 14. CICM Energy Overhead

Table 5. CICM Energy Overhead

	Energy Overhead [%]															
	CBC WtV				PMAC WtV				PMAC RBV				PMAC RBV Double			
Size [KB]	*1*	*2*	*4*	*8*	*1*	*2*	*4*	*8*	*1*	*2*	*4*	*8*	*1*	*2*	*4*	*8*
blowfish_enc	72.3	13.7	0.17	0.00	49.9	9.44	0.14	0.00	3.17	0.60	0.09	0.00	-12.2	-1.69	0.06	-0.01
cjpeg	25.3	5.22	1.13	0.29	17.5	3.59	0.77	0.19	1.17	0.24	0.00	0.00	-1.45	0.36	0.06	-0.00
djpeg	31.9	18.0	4.88	0.87	21.8	12.3	3.30	0.53	1.26	0.65	0.07	0.00	-2.43	-1.39	-0.21	-0.08
ecdhb	77.4	35.6	14.2	0.78	53.5	24.7	9.84	0.54	3.60	1.49	0.65	0.03	-11.8	-5.90	-1.94	-0.06
ecelgencb	68.7	18.8	6.57	0.34	47.5	13.0	4.55	0.24	3.24	0.78	0.30	0.01	-3.91	-3.07	-0.90	-0.04
ispell	99.6	88.4	49.6	10.4	68.6	60.8	34.3	7.21	4.43	4.09	2.64	0.51	-7.45	0.61	4.37	1.35
mpeg2enc	8.68	5.05	2.16	1.09	5.98	3.50	1.50	0.76	0.40	0.24	0.13	0.09	-0.56	-0.38	-0.05	-0.02
rijndael_enc	116.8	117.5	93.6	24.1	80.1	80.7	64.3	16.2	3.72	3.83	3.13	0.00	-14.0	-16.0	-8.23	7.61
stringsearch	96.3	79.2	23.6	10.4	66.5	54.9	16.3	7.14	4.44	3.85	1.38	0.31	2.02	-1.93	5.80	1.77
Total	**90.9**	**72.6**	**43.1**	**7.65**	**62.8**	**50.0**	**29.7**	**5.23**	**3.79**	**3.01**	**1.88**	**0.16**	**-9.17**	**-3.63**	**-0.12**	**1.54**

DICM. We also evaluate the energy overhead introduced by the data protection architecture. We use the most efficient CICM implementation, PMAC RbV, and evaluate the normalized values of energy consumed for the same architectures used for evaluating the CICM/DICM performance overhead. The normalized values of energy consumed are shown in Figure 15 and presented numerically in Table 6. These results show that protecting data integrity and confidentiality introduces much greater energy overhead than only protecting code. The bulk of this overhead comes from the sequence number cache. Due to this overhead, implementing the DICM architecture may not be appropriate for embedded systems with strict power constraints.

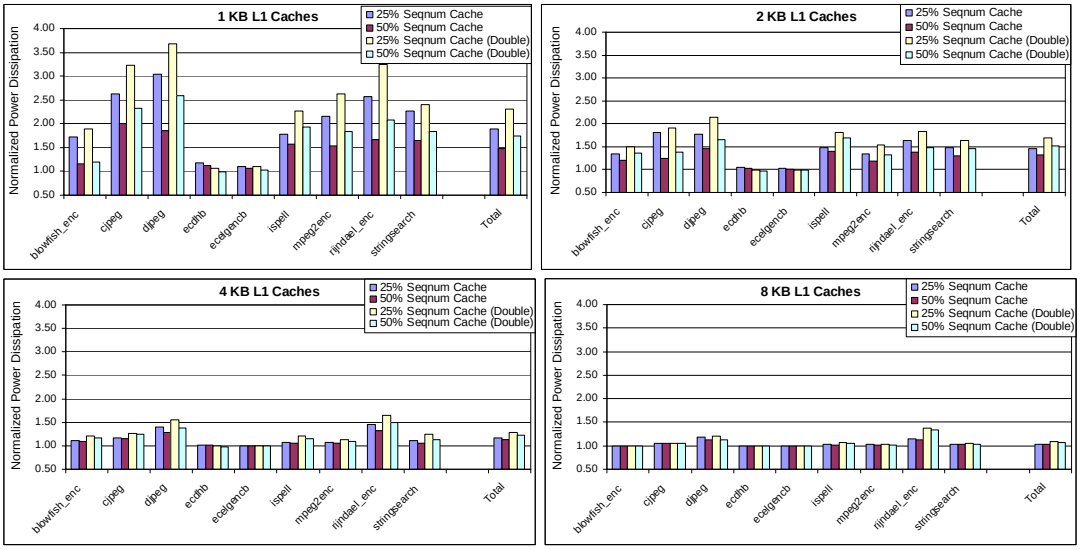


Figure 15. CICM/DICM Energy Overhead

Table 6. CICM/DICM Energy Overhead

	Energy Overhead [%]															
	Sequence Number Cache Size 25% of L1 Data Cache Size								Sequence Number Cache Size 50% of L1 Data Cache Size							
	Single Protected Block				*Double Protected Block*				*Single Protected Block*				*Double Protected Block*			
Size [KB]	*1*	*2*	*4*	*8*	*1*	*2*	*4*	*8*	*1*	*2*	*4*	*8*	*1*	*2*	*4*	*8*
blowfish_enc	72.5	33.9	10.4	0.03	89.41	49.4	21.7	0.03	16.2	20.5	9.2	0.03	20.2	35.7	16.9	0.02
cjpeg	163.6	81.1	17.8	5.05	222.5	91.0	26.9	5.69	101.3	24.6	15.2	4.48	133.2	38.6	23.9	4.92
djpeg	204.9	76.1	40.1	17.7	268.4	114.9	54.9	20.3	86.2	46.3	28.7	12.8	158.0	66.0	38.8	13.0
ecdhb	18.67	4.10	1.06	0.30	7.30	-1.44	-1.19	0.27	11.6	2.65	0.92	0.28	-0.53	-3.70	-1.41	0.24
ecelgencb	10.40	2.09	0.50	0.07	9.49	-0.49	-0.58	0.03	6.74	1.45	0.46	0.06	2.92	-1.87	-0.66	0.01
ispell	78.2	48.1	7.93	3.09	126.4	81.3	20.2	7.35	56.9	39.1	6.27	2.12	93.2	70.1	15.1	5.06
mpeg2enc	115.8	35.0	7.99	2.63	162.6	54.4	12.6	2.84	53.4	19.2	6.38	2.11	84.3	31.7	9.91	2.20
rijndael_enc	156.6	63.4	44.8	14.8	224.7	82.0	63.8	36.8	67.3	38.6	32.9	13.3	108.4	46.8	48.9	34.0
stringsearch	126.4	47.1	12.0	3.52	139.7	63.0	24.2	4.33	64.7	30.3	6.21	3.22	84.2	46.1	13.9	3.91
Total	**88.93**	**46.33**	**16.76**	**3.65**	**130.2**	**70.0**	**28.9**	**8.31**	**47.7**	**32.4**	**12.9**	**2.97**	**74.8**	**52.6**	**22.1**	**6.87**

6 Related Work

Proposals for computer security solutions fall into a broad spectrum ranging from predominantly software-centric to predominantly hardware-centric. Many approaches fall in between, requiring various levels of hardware and software support.

Software techniques may be classified as static or dynamic. Static software techniques rely on analysis of source code to detect security vulnerabilities. Dynamic software techniques require the augmentation of code or the operating system to detect attacks at runtime. Software techniques may not counter all attacks by themselves, as they lack generality, suffer from false positives and negatives, and often introduce prohibitive performance and power consumption overheads.

The trend of increasing transistor availability on microprocessor chips enables hardware approaches to the security problem. A number of hardware techniques have been proposed, varying in scope and complexity. Several proposals address specific vulnerabilities, such as the stack-smashing attack, by utilizing a secure hardware stack [33, 34] or encrypting address pointers [35], or randomizing the processor's instruction set [36]. The use of untrustworthy data for jump target addresses can be prevented by tagging all data coming from untrustworthy channels [37-39]; however, this approach requires relatively complex tracking of spurious data propagation and may produce false alarms.

The execute-only memory (XOM) architecture proposed by Lie *et al.* [40] provides an architecture meeting the requirements of integrity and confidentiality. Main memory is assumed to be insecure, so all data entering and leaving the processor while it is running in secure mode is encrypted. This architecture was vulnerable to replay attacks in its original form, but that vulnerability was corrected in the later work [4]. The drawbacks of this architecture are its complexity and performance overhead. XOM requires modifications to the processor core itself and to all caches, along with additional security hardware. This architecture also incurs a significant performance overhead of up to 50% by its designers' estimation.

The high overhead of XOM is reduced by the architectural improvements proposed by Yang *et al.* [41]. The authors only address confidentiality, as their improvements are designed to work with XOM, which already addresses integrity concerns. They propose to use a one-time pad (OTP) scheme for encryption and decryption, in which only the pad is encrypted and then XOR-ed with plaintext to produce ciphertext, or with ciphertext to produce plaintext. They augment data security by including a sequence number in the pad for data blocks, and require an additional on-chip cache for the sequence numbers. While their scheme greatly improves XOM's performance, it inherits its other weaknesses.

Suh and his colleagues [19] propose the AEGIS secure processor. They introduce physical unclonable functions (PUFs) to generate the secrets needed by their architecture. Memory is divided into four regions based on whether it is static or dynamic (read-only or read-write) and whether it is only verified or is both verified and confidential. They allow programs to change security modes at runtime, starting with a standard unsecured mode, then going back and forth between a mode supporting only integrity verification and a mode supporting both integrity and confidentiality. They also allow the secure modes to be temporarily suspended for library calls. This flexibility comes at a price; their architecture assumes extensive operating system and compiler support.

Milenković *et al.* introduced architectures for runtime verification of instruction block signatures [17]. Their architecture involves signing instruction blocks during a secure installation procedure. These signatures are calculated using instruction words, block starting addresses, and a secret processor key, and are stored together in a table in memory. At runtime, these signatures are recomputed and checked against signatures fetched from memory. The cryptographic function used in the architecture is a simple polynomial function implemented with multiple input shift registers. The architecture is updated in [7], adding AES encryption to increase cryptographic strength and embedding signatures with instruction blocks rather than storing them in a table. This architecture remains vulnerable to splicing attacks, since signatures in all programs use the same key. The vulnerability to splicing attacks and code confidentiality have been addressed in [42]. However, the proposed architecture only addresses instruction integrity and confidentiality and does not support any form of data protection.

Drinić and Kirovski [20] propose a similar architecture, but with greater cryptographic strength. They use a cipher block chaining message authentication code (CBC-MAC) cipher, and include the signatures in the cache line. They propose to reduce performance overhead by reordering basic blocks, so that instructions that may not be safely executed in a speculative manner are not issued until signature verification is complete. The drawback to this approach is that it requires significant compiler support, and may not consistently hide the verification overhead.

Furthermore, signatures are visible in the address space and stored in the cache, leading to cache pollution. Their architecture does not address confidentiality, and is vulnerable to replay and splicing attacks.

A joint research team from the Georgia Institute of Technology and North Carolina State University has proposed several secure processor designs. Yan et al. [27] describe a sign-and-verify architecture using Galois/Counter Mode cryptography. They protect dynamic data using split sequence numbers to reduce memory overhead and reduce the probability of a sequence number rollover. A tree-like structure is used to protect dynamic data against replay attacks. Rogers et al. [43] lower the overhead of the design by restricting the tree structure to only protect sequence numbers. They claim an average performance overhead of 11.9%. This overhead may be artificially low as they use "non-precise integrity verification," which allows potentially harmful instructions to execute and retire before they are verified. Their research has also been extended into the multiprocessor domain [44].

Several solutions have been developed by industry. Both Intel [45] and AMD [46] have implemented mechanisms that prohibit the execution of instructions from flagged areas of memory. IBM [47] and ARM [48] have developed security architectures to augment existing processor designs. Maxim proposes their DS5250 secure microprocessor [49] to be used as a secure coprocessor, handling sensitive operations while the primary processor remains untrusted.

This paper presents architectures offering improvements over the existing proposals. We offer lower complexity and performance overhead than XOM [40] and its derivatives [4, 41]. Unlike AEGIS [19] and the work presented in [20], our approach is almost completely hardware-centric, requiring no compiler support and only minimal operating system support. The proposed architectures are thus applicable to unmodified legacy code. Furthermore, we detect tampered instructions and data before they can cause harm to the system, unlike the architectures proposed in [27, 43-44]. Our approach offers high cryptographic strength with low performance and energy overhead.

7 Conclusions

This paper presents hardware security extensions suitable for implementation in embedded processors. With these extensions a program may run in an unprotected mode, code integrity only mode, code integrity and confidentiality mode, data integrity only mode, data integrity and confidentiality mode, or some combination of the above, depending on the desired level of security. To ensure code and data integrity, code and data blocks are protected by cryptographically sound signatures that are embedded in the code and data segments; these signatures are verified in runtime during program execution. Confidentiality is ensured by encrypting code and data blocks and signatures using a variant one-time-pad encryption scheme that induces minimal latency. To mitigate negative impacts of security extensions on performance and energy and memory requirements, we introduce and evaluate several architectural enhancements such as parallelizable signatures, speculative instruction execution, protecting multiple instruction and data blocks with a single signature, and sequence number caches. The experimental analysis shows that code protection introduces a very low complexity and negligible performance overhead. Data protection requires somewhat more complex hardware and increased performance overhead, but the total overhead for both code and data protection is less than 15%.

8 References

[1] J. Turley, "The Two Percent Solution," <http://www.embedded.com/story/OEG20021217S0039> (Available December, 2007).

[2] US-CERT, "Cert Statistics: Full Statistics," <http://www.cert.org/stats/fullstats.html> (Available December, 2007).

[3] BSA, "2007 Global Piracy Study," <http://w3.bsa.org/globalstudy/> (Available August, 2007).

[4] 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.

[5] 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.

[6] 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.

[7] M. Milenković, A. Milenković, and E. Jovanov, "Hardware Support for Code Integrity in Embedded Processors," in *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, San Francisco, CA, USA, 2005, pp. 55-65.

[8] 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.

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

[10] J. Black and P. Rogaway, "A Block-Cipher Mode of Operation for Parallelizable Message Authentication," in *Advances in Cryptology: Proceedings of EUROCRYPT 2002*, Amsterdam, Netherlands, 2002, pp. 384-397.

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

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

[13] P. Kocher, "Cryptanalysis of Diffie-Hellman, Rsa, Dss, and Other Systems Using Timing Attacks," 1995.

[14] P. Kocher, J. Jaffe, and B. Jun, "Differential Power Analysis," in *Advances in Cryptology: Proceedings of CRYPTO '99*, Santa Barbara, CA, USA, 1999, pp. 388-397.

[15] D. Boneh, R. A. DeMillo, and R. J. Lipton, "On the Importance of Checking Cryptographic Protocols for Faults," *Cryptology*, vol. 14, February 2001, pp. 101-119.

[16] O. Aciiçmez, Ç. K. Koç, and J.-P. Seifert, "On the Power of Simple Branch Prediction Analysis," in *Proceedings of the 2nd ACM Symposium on Information, Computer and Communications Security*, Singapore, 2007, pp. 312-320.

[17] M. Milenković, A. Milenković, 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.

[18] Y.-H. Wang, H.-G. Zhang, Z.-D. Shen, and K.-S. Li, "Thermal Noise Random Number Generator Based on Sha-2 (512)," in *Proceedings of 2005 International Conference on Machine Learning and Cybernetics*, Guangzhou, China, 2005, pp. 3970-3974.

[19] 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*, Madison, WI, USA, 2005, pp. 25-36.

[20] 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.

[21] J. H. An, Y. Dodis, and T. Rabin, "On the Security of Joint Signature and Encryption," in *Advances in Cryptology: Proceedings of EUROCRYPT 2002* Amsterdam, Netherlands, 2002, pp. 83-107.

[22] M. Bellare and C. Namprempre, "Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm," in *Advances in Cryptology: Proceedings of EUROCRYPT 2000*, Bruges, Belgium, 2000, pp. 531-545.

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

[24] 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.

[25] 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.

[26] 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.

[27] C. Yan, B. Rogers, D. Englender, Y. Solihin, and M. Prvulovic, "Improving Cost, Performance, and Security of Memory Encryption and Authentication," in *Proceedings of the 33rd annual international symposium on Computer Architecture*, Boston, MA, USA, 2006, pp. 179-190.

[28] 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.

[29] 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, pp. 3-14.

[30] 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.

[31] 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.

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

[33] 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.

[34] 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, pp. 50-56.

[35] 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.

[36] E. G. Barrantes, D. H. Ackley, S. Forrest, T. S. Palmer, D. Stefanović, and D. D. Zovi, "Randomized Instruction Set Emulation to Disrupt Binary Code Injection Attacks," in *Proceedings of the 10th ACM conference on Computer and communications security*, Washington, DC, USA, 2003, pp. 281-289.

[37] 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.

[38] 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.

[39] H. Chen, X. Wu, L. Yuan, B. Zang, P.-c. Yew, and F. T. Chong, "From Speculation to Security: Practical and Efficient Information Flow Tracking Using Speculative Hardware," in *Proceedings of the 35th International Symposium on Computer Architecture* Beijing, China, 2008, pp. 401-412.

[40] 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.

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

[42] A. Rogers, M. Milenković, and A. Milenković, "A Low Overhead Hardware Technique for Software Integrity and Confidentiality," in *International Conference on Computer Design*. Lake Tahoe, CA, USA, 2007.

[43] B. Rogers, S. Chhabra, Y. Solihin, and M. Prvulovic, "Using Address Independent Seed Encryption and Bonsai Merkle Trees to Make Secure Processors Os- and Performance-Friendly," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture* Chicago, IL, USA, 2007, pp. 183-196.

[44] B. Rogers, C. Yan, S. Chhabra, M. Prvulovic, and Y. Solihin, "Single-Level Integrity and Confidentiality Protection for Distributed Shared Memory Multiprocessors," North Carolina State University and Georgia Institute of Technology, 2008.

[45] Intel, "Execute Disable Bit and Enterprise Security," <http://www.intel.com/business/bss/infrastructure/security/xdbit.htm> (Available August, 2007).

[46] A. Zeichick, "Security Ahoy! Flying the Nx Flag on Windows and Amd64 to Stop Attacks," <http://developer.amd.com/articlex.jsp?id=143> (Available August, 2007).

[47] IBM, "Ibm Extends Enhanced Data Security to Consumer Electronics Products," <http://www-03.ibm.com/press/us/en/pressrelease/19527.wss> (Available August, 2007).

[48] ARM, "Trustzone: Integrating Hardware and Software Security Systems," <http://www.arm.com/pdfs/TZ_Whitepaper.pdf> (Available August, 2007).

[49] MAXIM, "Increasing System Security by Using the Ds5250 as a Secure Coprocessor," <http://www.maxim-ic.com/appnotes.cfm/appnote_number/3294> (Available August, 2007).

Austin Rogers is a graduate student in the Electrical and Computer Engineering Department at the University of Alabama in Huntsville, pursuing a PhD in computer engineering. Mr. Rogers' research interests include computer architecture and computer security. His personal interests include foreign languages and traditional music.

Aleksandar Milenković is an associate professor of Electrical and Computer Engineering at the University of Alabama in Huntsville. He currently directs the LaCASA Laboratory (http://www.ece.uah.edu/~lacasa/). His research interests include advanced architectures for the next generation computing devices, low-power VLSI, reconfigurable computing, embedded systems, and wireless sensor networks. Dr. Milenkovic received his Dipl. Ing., M.Sc., and Ph.D. degrees in Computer Engineering and Science from the University of Belgrade, Serbia, in 1994, 1997, and 1999, respectively.