An Implementation of Security Extensions for Data Integrity and Confidentiality in Soft-Core Processors

Austin Rogers[1], Aleksandar Milenković[2]
[1]Dynetics, Huntsville, Alabama
[2]ECE Department, The University of Alabama in Huntsville

Abstract-*An increasing number of embedded system solutions in space, military, and consumer electronics applications rely on processor cores inside reconfigurable logic devices. Ensuring data integrity and confidentiality is of the utmost importance in many such applications. This paper describes a practical implementation of security extensions for embedded systems built around soft-core processors. These extensions guarantee the integrity and confidentiality of sensitive data residing in external memory and prevent various types of physical attacks on systems working in adverse conditions. We describe the programming model, security architecture, and give an initial analysis of performance and complexity overheads caused by the security extensions.*

Keywords: Secure processors, data integrity and confidentiality, IP cores

1 Introduction

Embedded computer systems have become ubiquitous in modern society. We rely on them in a wide range of applications, from consumer electronics, communication, transportation, medicine, to national security. Many of these systems may operate in hostile environments where they are subjected to physical attacks aimed at subverting system operation, extracting key secrets, or intellectual property theft. Similarly, a system may operate in harsh conditions such as outer space, where natural phenomena may compromise the integrity of data. Security breaches in embedded systems could have wide ranging impacts, from loss of revenue to loss of life. These problems can be alleviated by building embedded systems that ensure (a) code and data integrity, thus preventing the execution of unauthorized instructions or the use of unauthorized data; and (b) code and data confidentiality, thus preventing the unauthorized copying of instructions or data [1].

A growing number of embedded systems are built around soft-core processors that are implemented on field-programmable gate array (FPGA) devices. This approach offers a number of advantages over traditional solutions and custom-designed processors because of its flexibility, platform independence, reduced cost, and immunity to obsolescence. The inherent flexibility of such systems allows us to rapidly prototype inexpensive security extensions.

This paper describes our approach to implementing security extensions to ensure data integrity and confidentiality in computer systems built around soft-core CPUs. This work represents our first foray into implementing security extensions in actual hardware. It builds on our existing theoretical work on secure processors [1, 2]. We focus on systems-on-a-chip implemented on FPGAs that store potentially sensitive information in off-chip memories. The chip is assumed to be inviolate, and thus any instructions or data contained in on-chip memory is assumed to be secure. We therefore focus on securing data that is stored off-chip in external memory and brought on-chip as needed, and do not address systems where instructions are stored off-chip. We take advantage of the inviolate nature of on-chip memory to keep the security hardware as simple as possible.

The design goals of this research are threefold. The first design goal is that the security extensions should ensure the integrity and confidentiality of data stored in off-chip memory. Protecting integrity means that the processor will only use authorized data; any tampering will be detected. Protecting confidentiality requires that sensitive data be encrypted and thus illegible by all unauthorized entities, computer or human. The second design goal is ease of use. The security extensions should be as transparent to the programmer as possible. The third and final design goal is that the security extensions introduce as little performance overhead as possible. Ideal security extensions would introduce minimal performance overhead. Additionally, the extensions should not require modification of any existing soft cores, as these are often distributed in binary-only formats.

We have implemented these security extensions in a system based on the Altera NIOS II processor. The test system was implemented on a Cyclone II FPGA using Altera's Quartus II toolchain. Our implementation ensures confidentiality by encrypting secure data that is stored off-chip. Integrity is ensured by signing that data using the cipher block chaining message authentication code (CBC-MAC) technique. The security extensions are transparent to the programmer other than requiring a function call to initialize the security-related hardware resources. We minimize performance overhead by overlapping cryptography with memory accesses and buffering verified blocks.

The remainder of this paper is organized as follows. Section 2 defines the threat models against which our security extensions defend. Section 3 describes how we achieve each of our design goals: security, transparency to the programmer, and as little performance overhead as possible. Section 4 evaluates our implementation with respect to complexity and

performance. Section 5 presents selected related work, and Section 6 concludes the paper.

2 Threat Models

Embedded computer systems may be subjected to a wide variety of attacks. Our focus is protecting data stored off-chip from physical attacks. These attacks require the attacker to have physical access to the system. The attacker can probe off-chip buses, and can observe and override bus transactions. Three possible physical attacks are spoofing, splicing, and replay.

A spoofing attack occurs when an attacker intercepts a request for a block of memory, and then manually supplies a block of his/her choice. In an unsecured system, the processor naïvely conducts a bus cycle, and is unaware that the data it received came from an attacker rather than from memory. The processor initiates a bus read cycle for a block at memory location A_i. The attacker intercepts the request and supplies a potentially malicious block M_i instead of the correct block A_i. A variation of the spoofing attack may occur in systems operating in adverse conditions, such as outer space, where external influences may corrupt data stored in memory.

Splicing attacks involve intercepting a request for a block of memory and then supplying the data from a valid, but different block. Once again, the unsecured processor is unaware that it has received an incorrect block from memory. The processor initiates a bus read cycle for a block at memory location A_i. The attacker intercepts the request and supplies a valid block from memory, but from address A_j rather than the desired address.

In a replay attack, the attacker intercepts a request for a block of memory, and then supplies an older copy of that block. The supplied block was correct at some point in the past, but it may have been superseded by a newer version. The processor initiates a bus read cycle for the data block at address A_i. The attacker intercepts the request and returns an older version of that block, which may be different from the current version in memory.

3 Implementing Security Extensions

This section describes how we achieve our three design goals. We begin with a description of how our design achieves security. We then discuss the programming model for our design, and the memory architecture necessary to implement it. We finally discuss how these security extensions are implemented in a hardware resource called the Encryption and Verification Unit (EVU).

3.1 Achieving Security

The basic unit of secure data is a protected block. In our implementation, we chose a protected block size of 32 bytes. In systems with on-chip caches, the cache block size, or some multiple thereof, is a convenient protected block size. For our initial implementation we do not use data caches.

Our design uses cryptography to protect the integrity and confidentiality of data stored off-chip. Confidentiality is protected by encryption. Integrity is protected by generating a 16-byte signature for each protected block of data. We defend against replay attacks by associating a sequence number with each protected block, and using it in encryption/decryption and signature calculation.

The confidentiality of data is protected by using a low-overhead variant of the one-time-pad encryption scheme. In this scheme, pads are calculated using Advanced Encrypted Standard (AES) ciphers, with the block address and sequence number as inputs. Equation (1) shows how this encryption is performed. The 32-byte plaintext data block D is divided into two 16-byte sub-blocks $D_{0:3}$ and $D_{4:7}$, which are separately encrypted to form ciphertext sub-blocks $C_{0:3}$ and $C_{4:7}$. *KEY1* is the 128-bit key used for pad generation. $A(SB_i)$ is the address of sub-block i, SN is the protected block's sequence number, and SP is a secure padding function that generates a unique 128-bit value from the 32-bit address and 32-bit sequence number.

$$C_{4i:4i+3} = D_{4i:4i+3} \; xor \; AES_{KEY1}(SP(A(SB_i), SN)) \quad (1)$$

Decryption is simply the reverse of this operation. The pads are calculated as in (1), and then XORed with the ciphertext sub-blocks to produce the desired plaintext sub-blocks.

Signatures are generated using the cipher block chaining message authentication code (CBC-MAC) method [3]. The protected block's signature S is calculated according to Equation (2). *KEY2* is another 128-bit key. SP is the secure padding function defined above, operating on the block's address $A(SB)$ and sequence number SN. The use of the block address prevents splicing attacks, the use of the block text prevents spoofing attacks, and the use of the sequence number prevents replay attacks. If the keys are generated randomly for each run, then cross-executable splicing attacks will also be prevented.

$$S = AES_{KEY2}[C_{4:7} \; xor \; AES_{KEY2}(C_{0:3} \; xor \; SP(A(SB), SN))] \quad (2)$$

If sequence numbers are stored off-chip, then they may be subjected to sophisticated replay attacks in which the sequence number is replayed as well as the protected block and its signature. This gives rise to the necessity of complex structures such as Merkle trees [4] to protect the sequence numbers. Our design assumes that sequence numbers are stored in on-chip memory and are thus invulnerable to replay attacks, and require no additional protection.

When the programmer reads from or writes to secure data at runtime, the appropriate sequence number, encrypted protected block, and signature are fetched. When the pads are available, the block is decrypted. As the two ciphertext sub-blocks become available, its signature is recalculated. If the calculated signature and fetched signature match, the block has not been subjected to tampering and the read or write operation can proceed. If the signatures do not match, a security violation has occurred and an interrupt is raised. More details are given below in Section 3.3.

In addition to preventing spoofing, splicing, and replay attacks, we must also prevent the programmer from inadvertently accessing uninitialized blocks. To that end, the sequence number value zero is reserved to indicate that its associated protected block is uninitialized. If a protected block's sequence number is zero, the programmer may write

to it, but not read from it. If the sequence number is nonzero, then the programmer may both read from and write to the protected block. A read from an uninitialized block will result in an interrupt.

Whenever a protected block is written back to main memory, its sequence number must be incremented and new pads calculated to encrypt the block. Sequence number overflows are undesirable, as they lead to pad re-use. Our design uses 32-bit sequence numbers; should a particular target application have a strong likelihood of a sequence number rollover, the design may be modified to use 64-bit sequence numbers.

In our design, the two cryptographic keys *KEY1* and *KEY2* are hard-coded in our security extension hardware. For greater security, they could be randomly generated at runtime for each application using methods such as physical unclonable functions [5]. In that case, these keys must be stored in the process control block in an encrypted form in the event of a context switch. An additional hard-coded internal key would be needed, which would then be used to encrypt these keys before and decrypt them after a context switch. Keys should never leave the chip in plaintext form. Hard-coded keys should only be used if the design will be protected by bitstream encryption.

3.2 Programming and Memory Model

An important design goal for these security extensions is that they be as transparent to the programmer as possible. To that end, our implementation does not require the programmer to use any special application programming interface (API) to read and store secure data. An initialization function must be called to initialize the necessary hardware resources (see Section 3.3 below). Thereafter, the programmer simply defines his or her pointers appropriately and uses them as normal.

This transparency is possible because of address mapping. A portion of the address space is set aside to physically store encrypted data. A similarly sized portion of the address space is mapped to the EVU. For instance, to read or write the n^{th} word of encrypted data, the programmer will read or write the n^{th} word in the EVU's address space. This transparency is illustrated in the code snippets in Figure 1. In the first snippet, OFFCHIP_MEM_BASE_ADDR defines the base address for off-chip memory. The second snippet accesses data relative to SECURE_DATA_BASE_ADDR, which defines the base address for accessing secure data via the EVU.

The memory architecture of our design is illustrated in Figure 2. The program text, heap, and stack are all stored in on-chip memory. Sequence numbers should also be stored on-chip. The figure depicts signatures as stored on-chip; they may also be stored in off-chip memory if desired. The shaded region in the address space contains the secure data in its encrypted form, which is physically stored off-chip.

The programmer may read data directly from the encrypted region, but the result would be a word of ciphertext. A direct write to this region would effectively constitute a spoofing attack, and would result in an interrupt the next time this secure data was properly accessed. Secure data should be accessed through an area of the address space assigned to the EVU. Addresses in this region are mapped to those in the encrypted data region, and the EVU handles all decryption and verification. If a block of secure data is no longer needed, its corresponding space in SDRAM may be reclaimed for unsecured use. However, that block must not be treated as secure data thereafter.

```
/* This code writes data directly to off-chip
   memory in an insecure manner. */
void Array_Access_Insecure()
{
   int i;
   int *pArray;

   pArray = OFFCHIP_MEM_BASE_ADDR;

   for(i = 0; i < 16; i++)
      pArray[i] = i;
}

/* This code writes secure data using the EVU. */
void Array_Access_Secure()
{
   int i;
   int *pArray;

   Initialize_EVU();

   pArray = SECURE_DATA_BASE_ADDR;

   for(i = 0; i < 16; i++)
      pArray[i] = i;
}
```

Figure 1 - Programmer's View of Securing Data in Off-Chip Memory

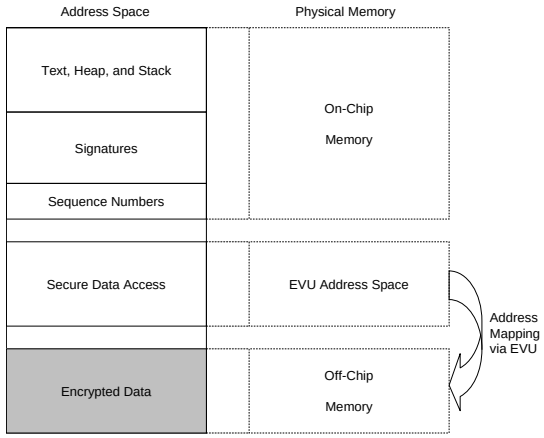

Figure 2 - Memory Architecture

The maximum number of 32-byte protected blocks is determined by the amount of memory allocated to storing signatures and sequence numbers. Each protected block requires a 16-byte signature and a 4-byte sequence number. Thus the maximum number of protected blocks N_{PB} in a system is limited by Equation (3). In this equation, $Sz(M_{sig})$ and $Sz(M_{seqnum})$ are the sizes in bytes of the memory regions allocated for storing signatures and sequence numbers, respectively.

$$N_{PB} = \min(Sz(M_{sig})/16, Sz(M_{seqnum})/4) \quad (3)$$

Since signatures introduce the greatest memory overhead, the designer may wish to fix the size of the region of memory allocated to signatures, and then calculate the required sizes

for the other memory regions. In our implementation, we chose to allocate four kilobytes of memory for storing signatures. This allows us to have 256 protected blocks of 32 bytes each, for a total of eight kilobytes of secure data. We thus require one kilobyte of on-chip memory for sequence numbers.

3.3 Implementation

The implementation of these security extensions must balance complexity and performance overhead, while at the same time not requiring the modification of any existing soft cores. To that end, the EVU is implemented as an on-chip peripheral attached to the bus. Other implementations are certainly possible, such as embedding the EVU functionality into a custom SDRAM controller. The implementation strategy we choose, however, allows our design to be flexible and applicable to existing systems.

Figure 3 shows a block diagram of our implementation of an embedded system incorporating our security extensions. All components of the baseline system are unshaded, while the shaded components are added to implement the security extensions. The baseline system for this implementation is a simple 32-bit NIOS II system-on-a-chip. On-chip memories are used to store program instructions and data (heap and stack). An SDRAM controller provides access to off-chip memory. The system is generated using Altera's System-on-a-Programmable Chip (SOPC) generator, part of the Quartus II toolchain. The on-chip bus interconnects conform to the Altera Avalon standard [6], with loads and stores occurring at the word level.

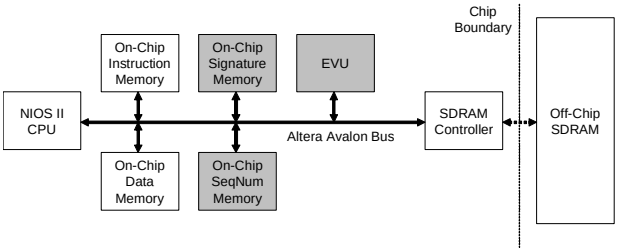

Figure 3 - System-on-a-Programmable-Chip Incorporating Security Extensions

The base system uses a simple NIOS II CPU with no data cache. In a NIOS II system with caches, cache lines are loaded and evicted via sequences of single-word accesses. The EVU would handle these like any other accesses.

The additional hardware to implement the security extensions consists of a discrete EVU peripheral, an on-chip memory for the sequence number table, and an on-chip memory for the signature table. Secure data is physically stored in its encrypted form in the off-chip SDRAM. (As mentioned earlier, signatures may also be stored off-chip if necessary.) The programmer may read directly from the SDRAM; however, if a location in the SDRAM containing secure data is read, encrypted data will be returned. SDRAM locations not used for storing secure data or signatures may be used to store non-sensitive plaintext data.

The internals and interfaces of the EVU are shown in Figure 4. In the upper left of this figure are the data and control registers for the EVU. Three data registers specify the base addresses of encrypted data in external memory, the signatures, and sequence numbers. These should be set in the aforementioned initialization function. The control register allows the programmer to reset the EVU and clear the interrupt. An Avalon bus slave interface allows access to these data and control registers.

A second Avalon bus slave interface is shown in the bottom left of the figure. This is the interface that the programmer will use to access secure data. Therefore, the portion of address space allocated to this interface should be commensurate with the amount of protected data. This is achieved by setting the width of the address signal on the slave interface. Avalon slave interface address signals are actually word indices rather than actual addresses. In our sample system, we have eight kilobytes of secure data, constituting 2,048 32-bit words. Thus, the address bus for this interface must be 11 bits wide to address all 2,048 words.

The memory access controller is a state machine responsible for fetching sequence numbers, signatures, and data blocks from memory and maintaining local buffers. The controller can access on-chip and external memories via an Avalon bus master interface. The EVU also contains an AES core and a state machine to control it. An interrupt interface allows interrupts to be raised by the memory access controller if the programmer tries to read from an uninitialized block or a fetched block and signature fails verification.

The upper right of the figure shows the various buffers used in the EVU. There are buffers for the fetched signature, calculated signature, the ciphertext block that has been read from memory or will be written to memory, the pads used to encrypt and decrypt the block, and the sequence number. An additional structure called the opportunity buffer attempts to reduce performance overhead by taking advantage of the locality of data accesses. Even though the processor will only read or write one word at a time, the entire protected block must be brought into the EVU in order to perform verification. This block is stored in the opportunity buffer as plaintext. Any further reads or writes to the protected block while it is buffered can be done within the EVU, without having to access memory. The block's address may be reconstructed

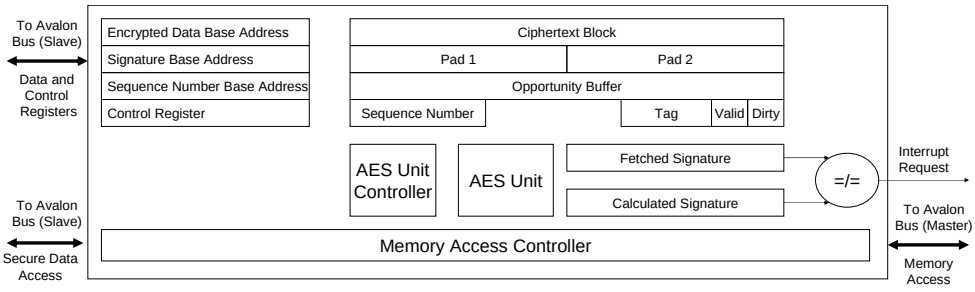

Figure 4 - Block Diagram of the Encryption and Verification Unit

from the opportunity buffer's tag. Its sequence number and the pads used to encrypt and decrypt it are also buffered.

When a word from a different block is requested, the block in the opportunity buffer must be evicted, along with its sequence number and signature. If the block is dirty, then it must be written back to external memory. The sequence number must be incremented and the pads recalculated before the plaintext block can be encrypted for storage. The opportunity buffer's tag is used to calculate the addresses for the block to be written back, its sequence number, and signature.

Figure 5 and Figure 6 list the algorithms used for reading and writing words of secure data, respectively. These algorithms reveal the latency hiding mechanisms used in the EVU. Whenever possible, cryptographic operations are done concurrently with memory operations to hide cryptographic latency. When writing to a protected block, new pads must be calculated once the sequence number has been incremented. As Figure 6 shows, the sequence number is only incremented when a block in the opportunity buffer is first marked dirty. Pad calculation is begun, and the processor is allowed to continue execution. If another read or write is initiated before the new pads have been calculated, the new access is stalled until the pads are completed.

```
Wait for any crypto operations from a previous access to complete.
Is buffer valid and does buffer tag match address?
  Yes: (read hit)
    Return word from buffer and exit.
  No: (read miss)
    Is buffer valid and dirty?
      Yes: (evict block from buffer)
        Encrypt block using buffered pads.
        Write sequence number and cryptotext block to memory.
        In parallel with memory write, calculate block signature.
        When signature is ready, write signature to memory.
        Continue with read miss operation.
      No: (do nothing, continue with read miss operation)
    Fetch sequence number from memory.
    Is sequence number nonzero?
      Yes: (block has been initialized)
        Read block and signature from memory.
        In parallel with memory accesses, calculate pads.
        Decrypt sub-blocks as pads and data are available.
        When block is fully available, calculate signature.
        Do calculated signature and fetched signature match?
          Yes: (everything is fine)
            Buffer block and pads; mark buffer valid and clean.
            Return word from buffer and exit.
          No: (security violation)
            Raise interrupt, mark buffer invalid, and exit.
      No: (trying to read an uninitialized block)
        Raise interrupt, mark buffer invalid, and exit.
```

Figure 5 - Algorithm for Secure Read

```
Wait for any crypto operations from a previous access to complete.
Is buffer valid and does buffer tag match address?
  Yes: (write hit)
    Latch word into buffer.
    Is buffer currently marked clean?
      Yes: (precompute pads for eventual writeback)
        Mark buffer dirty.
        Increment buffered sequence number.
        Start calculation for new pads, and exit.
      No: (do nothing, exit)
  No: (write miss)
    Is buffer valid and dirty?
      Yes: (evict block from buffer)
        Encrypt block using buffered pads.
        Write sequence number and cryptotext block to memory.
        In parallel with memory write, calculate block signature.
        When signature is ready, write signature to memory.
        Continue with write miss operation.
      No: (do nothing, continue with write miss operation)
    Fetch sequence number from memory.
    Is sequence number nonzero?
      Yes: (block has been initialized)
        Read block and signature from memory.
        In parallel with memory accesses, calculate pads.
        Decrypt sub-blocks as pads and data are available.
        When block is fully available, calculate signature.
        Do calculated signature and fetched signature match?
          Yes: (everything is fine)
            Buffer block and pads; mark buffer valid and dirty.
            Increment sequence number.
            Latch word into buffer.
            Start calculation for new pads, and exit.
          No: (security violation)
            Raise an interrupt, mark buffer invalid, and exit.
      No: (initialize the block)
        Set sequence number to 1.
        Start pad calculation.
        Load buffer with zeros; mark buffer valid and dirty.
        Set block init bit.
        Latch word into buffer and exit.
```

Figure 6 - Algorithm for Secure Write

4 Evaluation

The implementation of our security extensions was synthesized, placed, routed, and deployed on a Terasic DE2-70 [7], a low-cost development and education board. The DE2-70 includes an Altera Cyclone II 2C70 FPGA. The system was then evaluated for complexity and performance.

4.1 Complexity

Three discrete components were added to the baseline system to implement the security extensions: the EVU, a 1 KB on-chip memory for the sequence number table, and a 4 KB on-chip memory for the signature table. The complexity overhead introduced by these components is shown in

Table 1. The figures in the table are reported by the Quartus II tool. The overhead introduced by the AES core is shown separately from that of the EVU, as it contributes nearly half the additional logic. The AES core used in this implementation is an open-source intellectual property (IP) core, requiring 15 clock cycles per operation [8]. It is not pipelined.

Table 1 - Complexity Overhead

Component Name	Logic Cells	Dedicated Logic Registers	M4K Blocks
EVU	3,290	1,910	0
AES Core	5,031	658	0
Sequence Number Memory (1 KB)	2	0	2
Signature Memory (4 KB)	2	0	9
Total Overhead:	8,325	2,568	11

The EVU itself requires many registers to implement the opportunity buffer. The additional memories consume little in the way of logic cells, but do consume M4K blocks, which are on-chip RAM resources. Recall that signatures need not be stored on-chip; they may be stored in an off-chip memory if on-chip memory space is at a premium. The base system (without the EVU and additional memories) took approximately 7% of our target FPGA's resources. The secure system required 21%, three times the total on-chip resources.

4.2 Performance

The performance overhead introduced by the security extensions was evaluated by running a microbenchmark to stress-test the system. The microbenchmark potentially introduces far greater overhead than an actual application. It reads and writes to an array in memory with a varying stride factor. When performing write accesses, a miss in the opportunity buffer will always cause a writeback. Baseline results are measured by reading and writing directly to SRAM. Overhead is determined by reading and writing using the secure extensions. The array is read or written many thousands of times, and the total number of clock cycles required for all accesses is counted. This value is then divided by the total number of accesses to provide the average number of clock cycles per access. Varying the stride factor allows the benchmark to vary the degree to which it takes advantage of the opportunity buffer. With a stride of one, it takes full advantage of the buffer, with an opportunity buffer miss every eighth access. With a stride of eight, an opportunity buffer miss occurs every access, thus allowing us to measure the average time required to fetch and verify a protected block from off-chip memory. Neither the baseline nor secure systems contain data caches or any other performance enhancement mechanisms. This allows us to see the worst-case, bottom-line latencies. Therefore, the latencies reported in this section are worse than they would be in a more realistic system containing one or more levels of data cache.

The microbenchmark was run with signatures stored on-chip and with signatures stored in off-chip SDRAM. The results of these runs are shown in Table 2. Results are shown for the baseline system (reading or writing directly to or from off-chip SDRAM) and for the secure system taking advantage of the opportunity buffer to varying degrees.

This table shows that a read miss in the opportunity buffer introduces 59 cycles of overhead, regardless of whether signatures are stored on-chip or off-chip. When the benchmark takes advantage of the opportunity buffer, performance increases, even to the point of performing better than the baseline system when a read miss occurs every eighth access (due to a pre-fetching effect). Write misses, however, introduce much higher overheads. This is because the EVU stalls the write transaction while performing the writeback operation (if necessary) and then fetching and verifying the protected block. In the baseline case, the SDRAM controller buffers the block and does not stall the transaction. The overhead introduced by the writeback can be found by subtracting the number of cycles reported for a read miss by that reported for a write miss. This shows that a writeback takes 60 cycles when signatures are stored on-chip, and 69 cycles when signatures are stored off-chip.

Table 2 – Performance Overhead, Signatures Stored On-Chip/Off-Chip

	SIG. ON-CHIP		SIG.OFF-CHIP	
	Avg Cycles	Over-head	Avg Cycles	Over-head
Read Accesses				
Baseline System	24	1	24	1
Secure, Miss Every 8th Access	19	0.79	19	0.79
Secure, Miss Every 4th Access	28	1.17	28	1.17
Secure, Miss Every Other Access	46	1.92	46	1.92
Secure, Miss Every Access	83	3.46	83	3.46
Write Accesses with Writebacks				
Baseline System	2	1	2	1
Secure, Miss Every 8th Access	20	10	21	10.5
Secure, Miss Every 4th Accesses	38	19	40	20
Secure, Miss Every Other Access	73	36.5	77	38.5
Secure, Miss Every Access	143	71.5	152	76

The major contributors to performance overhead were identified using built-in counters inside the EVU. The counters reported that a read miss in the opportunity buffer takes 75 clock cycles. Further analysis revealed that memory accesses completed long before the cryptographic operations, as depicted in Figure 7. Latency from cryptographic operations dominates, thus explaining why the overhead on a read miss is not dependent on whether or not signatures are stored on- or off-chip. This suggests that performance could be improved by using either a pipelined AES core or two AES cores operating in parallel. Either of those arrangements would also allow signatures to be generated using the parallel message authentication code (PMAC) technique [1], which will further decrease performance overhead.

In addition to the microbenchmark, an actual benchmark was ported to run on the secure system. The Rijndael benchmark from the MiBench suite [9] was modified to read its keys and input data from, and store its outputs to, secure memory. The performance overhead was found to be only 1.01 times (about 1%) for both on-chip and off-chip signatures. This indicates that actual applications should exhibit far less overhead than the stress-test microbenchmark.

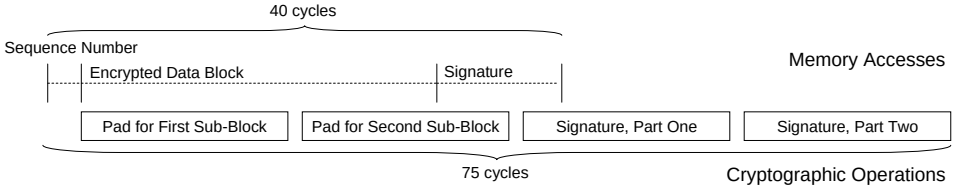

Figure 7 - Performance Overhead on a Read Miss (Not to Scale)

5 Related Work

Several computer security researchers have targeted the reconfigurable computing domain. In this section, we briefly survey several existing proposals for implementing security extensions in reconfigurable logic.

Wang, Yeh, Huang, and Wu [10] developed a cryptographic coprocessor on an FPGA to accelerate cryptographic functions in an embedded system. Zambreno, Honbo, Choudhary, Simha, and Narahari [11] propose to use an FPGA as an intermediary, analyzing all instructions fetched by a processor. It calculates checksums for basic blocks using two different methods, such as a hash on the code and the list of registers used by instructions, and compares the two checksums at the end of the basic block. The level of security provided by this approach is an open question, and requires extensive compiler support, including the insertion of dummy instructions, to establish the appropriate "register stream." This leads to a rather high overhead of around 20%, and only supports instruction integrity and confidentiality (by means of optional encryption).

Suh, Charles, Ishan, and Srinivas [5] 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.

Although several research efforts have developed security solutions for microprocessors on FPGAs, our research is unique in that it presents practical security extensions that can be implemented non-invasively in soft-core-based systems on inexpensive FPGAs. Reconfigurable devices are excellent platforms for this research, allowing us to rapidly prototype new designs and evaluate trade-offs.

6 Conclusions

This paper has shown one possible implementation of security extensions ensuring data integrity and confidentiality in embedded systems utilizing soft-core CPUs. This is our first cut at implementing security extensions in actual hardware, and will be improved in future work. Possible avenues for further work include extending this design to protect the integrity and confidentiality of instructions as well as data, and implementing the EVU in a more complex system containing instruction and data caches. The design could also be tested with different AES units, exploring the tradeoffs between using a single non-pipelined AES unit, multiple AES units, or a single pipelined unit.

7 References

[1] 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.

[2] A. Milenković, M. Milenković, and E. Jovanov, "An Efficient Runtime Instruction Block Verification for Secure Embedded Systems," *Journal of Embedded Computing*, vol. 4, January 2006, pp. 57-76.

[3] 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.

[4] 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, CA, USA, 2003, pp. 295-306.

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

[6] Altera, "Avalon Interface Specifications," <http://www.altera.com/literature/manual/mnl_avalon_spec.pdf> (Available January, 2009).

[7] T. Technologies, "Altera De2-70 - Development and Education Board," <http://www.terasic.com.tw/cgi-bin/page/archive.pl?Language=English&CategoryNo=39&No=226> (Available September, 2008).

[8] H. Satyanarayana, "Aes128," <http://www.opencores.org/projects.cgi/web/aes_crypto_core/> (Available August, 2008).

[9] 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.

[10] C.-H. Wang, J.-C. Yeh, C.-T. Huang, and C.-W. Wu, "Scalable Security Processor Design and Its Implementation," in *Asian Solid-State Circuits Conference*, Hsinchu, China, 2005, pp. 513-516.

[11] J. Zambreno, D. Honbo, A. Choudhary, R. Simha, and B. Narahari, "High-Performance Software Protection Using Reconfigurable Architectures," *Proceedings of the IEEE*, vol. 94, February 2006, pp. 419-431.