Memory Security Management for Reconfigurable Embedded Systems

Romain Vaslin, Guy Gogniat, Jean-Philippe Diguet
European University of Brittany
UBS - Lab-STICC
Lorient, 56100, FRANCE
vaslin@univ-ubs.fr

Russell Tessier, Deepak Unnikrishnan
University of Massachusetts
ECE department
Amherst, MA 01003, USA
tessier@ecs.umass.edu

Kris Gaj
University of George Mason
ECE department
Fairfax, VA 22030, USA
kgaj@gmu.edu

Abstract

The constrained operating environments of many FPGA-based embedded systems require flexible security that can be configured to minimize the impact on FPGA area and power consumption. In this paper, a security approach for external memory in FPGA-based embedded systems that exploits FPGA configurability is presented. Our FPGA-based security core provides both confidentiality and integrity for data stored externally to an FPGA which is accessed by a processor on the FPGA chip. The benefits of our security core are demonstrated using four embedded applications implemented on a Stratix II device. Each application requires a collection of tasks with varying memory security requirements. Our security core is used in conjunction with a NIOS II soft processor running the MicroC/OS II operating system. An average memory and energy savings of about 64% and 16%, respectively, is achieved for the four applications versus a non-configurable, uniform security approach.

1 Introduction

FPGAs are quickly becoming ubiquitous components in many low-cost embedded systems. These systems often contain little more than an FPGA, external memory, I/O ports, and interfaces to sensors and monitors. In addition to standard concerns regarding system performance and power consumption, security has become a leading issue for many embedded applications. Although the programmed contents of FPGAs are frequently protected with bitstream encryption [13], external memory transfers can easily be observed and may reveal important application information. Some memory protection can be provided by simply encrypting data prior to external memory transfer. Although data encryption techniques are widely known and used, simply modifying the values of data and instructions is generally thought to be insufficient to provide full protection against information leakage [2].

An important aspect of FPGA-based systems that distinguishes them from their ASIC-based counterparts is their ability to exactly meet designer hardware requirements on a per application basis. This flexibility has been shown to reduce application power consumption and improve system performance for a variety of FPGA-based systems [10]. However, the implementation of specific security policies in FPGAs [3] has only recently received attention. As FPGA use in embedded systems grows, the need to efficiently implement security protocols that can be easily updated becomes important.

In this paper, we describe a new FPGA-based approach which provides security to off-chip instruction and data accesses made by a processor on the FPGA chip (soft or hard core). A hardware-based security core determines the appropriate data security level as memory accesses occur in conjunction with an embedded real-time operating system. Our approach allows for the optimization of security core size on a per application basis based on memory footprint and security level requirements. The implementation has been parameterized to allow for straightforward adaptation to a variety of applications. To demonstrate the effectiveness of our approach, we implement four versions of the core with four different multi-task applications, each requiring different mixes of security levels. These designs have been tested using an Altera NIOS II soft processor [1] on a Stratix II based prototyping board. The NIOS II runs the MicroC/OS II operating system [5] to schedule tasks for each application. The new approach is optimized for FPGAs, which allow secure dynamic reconfiguration via an encrypted bitstream versus ASICs which have a fixed configuration.

The paper is organized as follows. Section 2 describes data security issues for embedded systems and previous approaches to address memory protection. Section 3 provides details of our security core. In Section 4 we describe the integration of our core with NIOS II and its use with four embedded applications. A description and analysis of experimental results are provided in Section 5. Section 6 concludes the paper.

	PE-ICE [4]	XOM [6]	AEGIS [8]
Software execution loss	50%	>50%	>50%
External memory increase	50%	50%	>50%
Security level	$1/2^{32}$	$1/2^{128}$ or $1/2^{160}$	$1/2^{160}$

Table 1: Security and performance levels for memory protection

2 Background

2.1 Embedded System Memory Threats

The external memory of an embedded system can face a variety of attacks [4] resulting from either the probing of the interface between a processor and the memory or physical attacks on the memory itself (fault injection). Bus probing results in the collection of address and data values which can be used to uncover processor behavior. The encryption of data values using algorithms such as the Advanced Encryption Standard (AES) or Triple Data Encryption Standard (3DES) prior to their external transfer guarantees data *confidentiality*. Data encrypted with these algorithms cannot be retrieved without the associated key. However, even encrypted data and their associated addresses leave memory values vulnerable to attack. Well-known attacks [4] include spoofing, relocation, and replay attacks. A *spoofing* attack occurs when an attacker places a random data value in memory, causing the system to malfunction. A *relocation* attack occurs when a valid data value is copied to one or more additional memory locations. A *replay* attack occurs when a data value which was previously stored in a memory location is substituted for a new data value which overwrote the old location. For all three cases, it may be possible for an embedded system to successfully decrypt data, but system behavior will be negatively impacted. Processor instructions are particularly vulnerable to relocation attacks since specific instruction sequences can be repeated in an effort to force a system to a specific state. Specific approaches that maintain the *integrity* of data from these types of attacks are needed to secure embedded systems. Integrity is a mechanism which guarantees that a data value has not been altered during storage or transfer. In previous works and in our work, attacks such as side channel or fault injection are not taken in account by our threat model. Several papers already deal with these kind of issues.

2.2 Related Work

A number of techniques have been developed that provide data confidentiality and integrity in processor-based systems. For these systems [4] [6] [8] [9], confidentiality is provided via data encryption using AES or 3DES. Data is encrypted prior to off-chip transfer and decrypted following data retrieval from memory. Data integrity is typically maintained by hashing data values in a hierachical fashion [4] [6] [9] [9]. The *security level* of these schemes measures the likelihood that an attacker could break the provided integrity using a changed data value that could pass the integrity check. Even though these solutions have been shown to be effective, the cost of security can be high in terms of secure off-chip memory space needed to store hash and tag values for each data item and increased read latency due to integrity checking. As summarized in Table 1, performance and off-chip memory overheads involved can often be 50% or higher, constraining embedded systems. On-chip memory usage for these approaches is not significant.

A distinguishing feature of our new low-overhead approach is its ability to offer configurable data security levels for different tasks in the same application. The confidentiality and integrity approach in AEGIS [9] most closely matches our approach. AEGIS also allows for the selection of different security levels for different portions of memory. This memory mapping must be identified during compilation and new instructions are added to the processor for operating system use. These instructions take advantage of hardware security primitives to enter and access OS kernel primitives. Overall, this approach adds complexity to both the processor architecture and the operating system. Although it appears that other data confidentiality and integrity approaches [4] [6] could be easily extended to multiple tasks, selective security based on memory addresses has not been reported for them.

The scope of our work is limited by the same threat model assumed by the earlier approaches. Our approach targets replay, relocation, and spoofing attacks. We assume that the FPGA is physically secure from attack and has been properly configured.

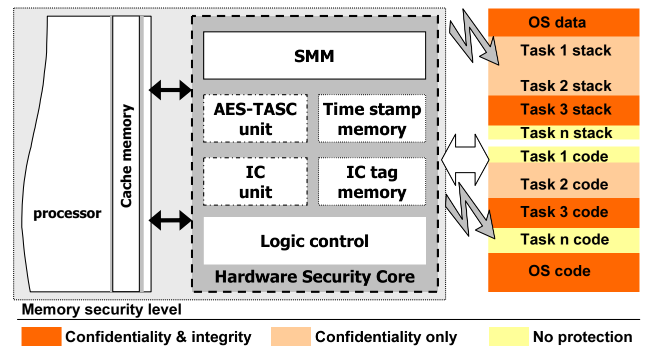

Figure 1: Memory security system overview

3 Memory Security Architecture

Our approach, shown in Figure 1, relies on a hardware security core (HSC) fashioned from FPGA logic and embedded memory which is able to manage different security levels depending on the data address received from the processor. A small lookup table (the security memory map or SMM) is included in the core to store the security level of memory segments accessed by tasks. Three security levels are possible for each memory segment: confidentiality-only, confidentiality and integrity, or no security. The implementation of the security policy in the SMM is indepen-

dent of the processor and associated operating system. The configuration of the SMM and the rest of the core is contained in the encrypted FPGA bitstream. The isolation of the SMM makes it secure from software modification at the expense of software-level flexibility. New multi-task applications require a new FPGA bitstream to achieve a new memory security protocol.

3.1 Security Level Management

The increased use of soft and hard-core processors in FPGAs has facilitated the use of operating systems in FPGA-based systems. The use of an OS provides a natural partitioning of application code and data. Based on this partitioning, developers can specify a desired security partitioning for each. In Figure 1, the application instructions and stack data of Task 1 have different security levels. In this case, the application designer may wish to keep task processor data secure to prevent copying. The application code may be less sensitive, eliminating a need for security. Our approach is designed to be used in conjunction with a memory management unit (MMU). This unit ensures that a task will not read or write memory segments that are not associated with it, creating a security risk if security levels differ. The availability of configurable security levels provides a benefit over requiring all memory to perform at the highest security level of confidentiality and integrity checking. The amount of on-chip memory required to store tags for integrity checking can be reduced if only external memory that requires security is protected. Additionally, the latency and dynamic power of unprotected memory accesses is minimized since unneeded security processing is avoided. FPGA reconfigurability allows for the optimization of the required on-chip storage and modification via a new bitstream.

3.2 Memory Security Core Architecture

Our FPGA core for management of memory security levels is an extension of a preliminary version [11] [12] which provides uniform security for all tasks and memory segments and uses one-time pad (OTP) operations [8] for confidentiality and cyclic redundancy checks (CRC) for integrity checking. The linearity of CRC operations has been found to expose data to integrity check failures in some cases. Confidentiality in our new system is similar to the AES-based encryption scheme called Binary Additive Stream Cipher [7]. Rather than encrypting write data directly, our approach first generates a *keystream* using AES, which operates using a secret key stored in the FPGA bitstream. In our implementation, a time stamp value, the data address, and the segment ID of the write data are used as input to an AES encryption circuit to generate the keystream. This keystream is then XORed with the data to generate ciphertext which can be transferred outside the FPGA. The timestamp is incremented during each cache line write. The same segment ID is used for all cache lines belonging to a particular application segment. The benefit of the AES-TASC (AES in time address segment counter mode) approach versus direct data encryption of the write data can be seen during data reads. The keystream generation can start immediately after the read address is known for read accesses. After the data is retrieved, a simple, fast XOR operation is needed to recover the plaintext. If direct data encryption was used, the decryption process would require many FPGA cycles after the encrypted data arrives at the FPGA. Thus, the use of AES-TASC significantly reduces the read latency of security. One limitation of this approach is a need to store the time stamp (TS) values for each data value (usually a cache line) in on-chip storage so it can be used later to verify data reads. A high-level view of the placement of security blocks is seen in Figure 1.

A step-by-step description of AES-TASC protected data reads and writes are shown in Algorithm 1 (steps 2-4) and Algorithm 2 (steps 1, 3, 5). From a security standpoint, it is essential that the keystream applied to encrypt data is used only one time. Since the keystream is obtained with AES, the AES inputs also need to be used just one time. If the same keystream is used several times, information leakage may occur since an attacker may be able to determine if data encrypted with the same keystream have the same values. The use of the data memory address in the generation of the keystream (Figure 2) protects the data value from relocation attacks. To prevent replay attacks, a simple time stamp generator, such as a counter, is used. As shown in Algorithm 1, the TS value associated with each data address is incremented by 1 after each write to the memory. For each new cache line memory write request, the system will compute a different keystream since the value of TS is incremented. During a read, the original TS value is used for comparative purposes (Algorithm 2). The retrieved TS value is provided to AES during the read request. The AES result will give the same keystream as the one produced for the write request and the encrypted data will become plaintext after being XORed (Algorithm 2, step 5).

Algorithm 1 - ***Cache memory write request:***

$1 - Integrity\ tag\ computation : IC\ tag = IC_AES\{plaintext\}$
$2 - Time\ stamp\ incrementation : TS = TS + 1$
$3 - Keystream = AES\{TS, address, Segment\ ID\}$
$4 - Ciphertext = plaintext \oplus keystream$
$5 - Ciphertext \Rightarrow external\ memory$
$6 - Time\ stamp\ storage : TS \Rightarrow TS\ memory$
$7 - Integrity\ tag\ storage : IC\ tag \Rightarrow IC\ memory$

Read-only data, such as processor instructions, do not

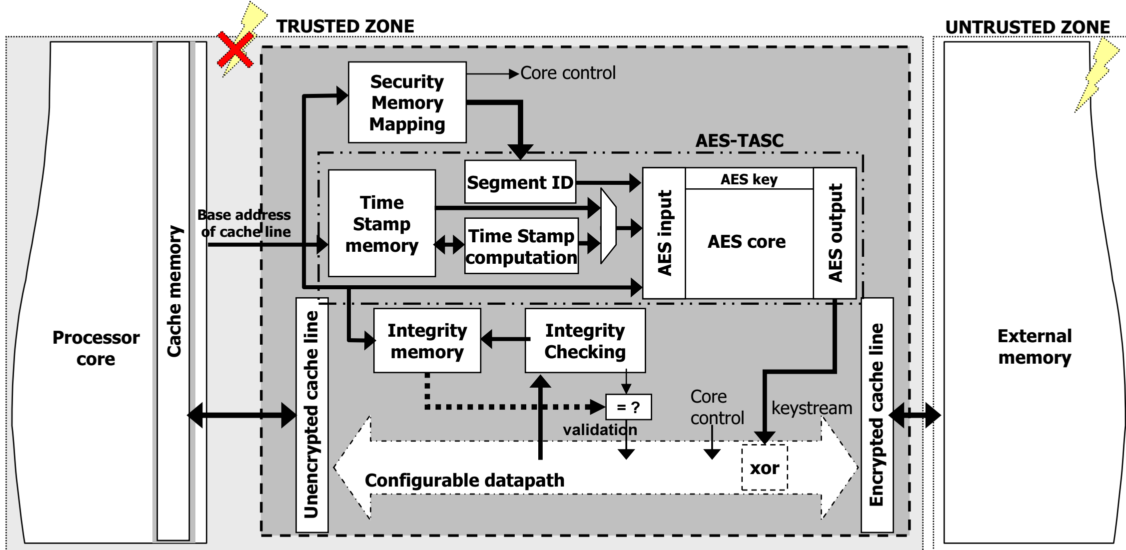

Figure 2: Hardware Security Core architecture

require protection from replay attacks because these data are never modified. No TS values are needed for these data so the amount of on-chip TS memory space can be reduced accordingly. Read-only data may be the target of relocation attacks but the address used to compute the AES-TASC guarantees protection against these attacks. The use of time stamps and data addresses for AES-TASC protects read/write data against replay and relocation attacks. If a data value is replayed, the TS used for ciphering will differ from the one used for deciphering. If a data value is relocated, its address will differ from the one used to generate the keystream. In both cases, the deciphered data will be invalid.

Algorithm 2 - ***Cache memory read request:***

$1 - TS\ loading : TS \Leftarrow TS\ memory$
$2 - IC\ tag\ loading : IC\ tag \Leftarrow IC\ tag\ memory$
$3 - Keystream = AES\,\{TS, address, Segment\ ID\}$
$4 - Ciphertext\ loading : Ciphertext \Leftarrow external\ memory$
$5 - Deciphered\ data = Ciphertext \oplus keystream$
$6 - Integrity\ checking : IC\ tag \equiv IC_AES\,\{Deciphered\ data\}$
$7 - Deciphered\ data \Rightarrow cache\ memory$

The need for unique time stamps creates a problem if the time stamp generation counter rolls over and starts reusing previously-issued time stamps. A typical solution to this issue involves the reencryption of stored data with new time stamps [14]. Recently, an AES-TASC solution which uses multiple time stamp generator counters [14] was proposed. If a time stamp counter reaches its maximum value, only about half the data must be reencrypted. With our security approach, the same idea can be applied based on segment IDs. If a timestamp associated with a segment rolls over, the segment ID value is incremented. All the data included in the segment are reencrypted with the new segment ID value. The use of the segment ID in keystream generation helps avoid the issue of matching time stamp values in this case. If reencryption due to counter rollover is needed, only a portion of the external memory is affected. This feature can help reduce the down time of an embedded system.

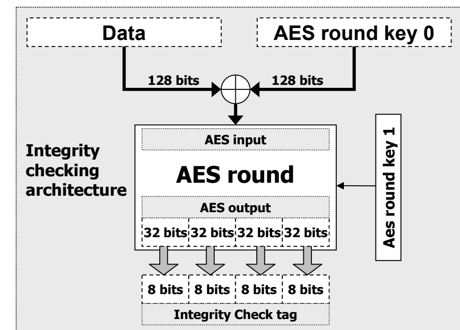

Figure 3: Integrity checking architecture

The memory security core must be able to detect that a deciphered data value is correct following a memory read. Our core includes an extension to AES-TASC encryption that evaluates data integrity. As shown in Figure 3, the integrity checking unit is composed of one AES round. The integrity check (IC) tag is composed of bits from the AES-round output. These output bits are selected in such a way that if any data input bit changes, multiple output bits are affected.

The IC tag of the cache line to be encrypted (step 1 in Al-

	Confidentiality and Integrity						Confidentiality						No protection					
App.	Code			Data			Code			Data			Code			Data		
	kB	Tasks	Segs	kB	Tasks	Segs	kB	Tasks	Segs	kB	Tasks	Segs	kB	Tasks	Segs	kB	Tasks	Segs
Image	25	2	5	33	2	3	7	2	1	10	2	1	38	1	1	16	1	1
VOD	26	5	3	113	6	4	58	1	1	0	0	0	68	1	1	318	1	1
Comm	71	6	1	28	0	2	0	0	0	40	6	1	0	0	0	0	0	0
Hash	0	0	0	0	0	0	92	5	1	0	0	0	0	0	0	55	5	1

Table 2: Application memory protection details by protection level

gorithm 1) is stored in the IC memory (step 7 in Algorithm 1) prior to keystream generation. Later, when the processor core requests a read, the IC tag result of the final XOR operation is compared with the integrity check value stored in the memory (step 6 in Algorithm 2). If data is changed following storage by a replay or relocation attack, the IC tag of the retrieved value will differ from the stored value, so the attack is detected.

The storage required for integrity check values impacts the security level provided. In this case, a 128 bit data input and a 32 bit IC tag are used. As each cache line is composed of 256 bits, an attacker only has a 1 out of 2^{64} probability of successfully modifying the encrypted value and achieving the original IC tag value. This security level is similar to values for previous area-intensive approaches shown in Table 1, which are appropriate for FPGAs. As shown in Figure 2, the data paths in the security core are controlled by the SMM output. The SMM receives the base address of the cache line and sends signals to configure the correct data path inside the core.

4 Experimental Approach

In order to validate the benefits of our approach, an architecture based on an Altera NIOS II soft processor [1] was developed. Our security core and associated memory was implemented in FPGA logic and embedded memory and interfaced to the processor via an Avalon bus. In separate sets of experiments, the NIOS II was first allocated instruction and data caches of 2 KB bytes and then 512 bytes. The current implementation of NIOS II does not use an MMU. The widely-used MicroC/OS-II [5] embedded operating system was used to validate our approach. MicroC/OS-II is a scalable, preemptive and multitasking kernel. The OS can be configured by the designer during application design to use only the OS features that are needed. A priority-based scheduling is used to evaluate which one of up to 64 tasks runs at a specific point in time. MicroC/OS-II uses a hardware timer to produce ticks which force the scheduler to run.

To explore the impact of the security management on performance, area, and power consumption, 4 multi-task applications were used. These applications include:

- **Image processing** - This application selects one of two values for a pixel and combines the pixels into shapes. Pixel groups that are too small are removed from the image. This process is called morphological image processing.

Application	Tasks	Mem Segs.	Total mem (kB)	
			Code	Data
Image	5	12	80	59
VOD	7	10	152	431
Comm	6	4	71	68
Hash	5	2	92	55

Table 4: Application memory overview

- **Video on demand (VOD)** - This application includes a sequence of operations needed to receive transmitted encrypted video signals. Specific operations include Reed Solomon (RS) decoding, AES decryption, and MPEG-2 decoding with artifact correction.

- **Communications** - This application includes a series of tasks needed to send and receive digital data. Specific operations include Reed Solomon decoding, AES encryption, and Reed Solomon encoding.

- **Hash** - This application can perform selective hashing based on a number of common algorithms. Supported hash algorithms include MD5, SHA-1 and SHA-2.

Each of these applications can benefit from a selective memory security policy that can be uniquely implemented in separate FPGA bitstreams. For the image processing application, image data and application code used to filter data is protected, but data and code used to transfer information to and from the system is not. For the VOD application, deciphered image data and AES specific information (e.g. the encryption key) is considered critical. Also, the MPEG algorithm is considered proprietary and its source code is encrypted, while MPEG data and RS code and data are left unprotected. For the communications application, all data is considered sensitive and worthy of protection. In order to guarantee correct operation, the code must not be changed, so confidentiality and integrity checking is applied to all code. Application data is only protected for confidentiality. Hash application code is only encrypted (confidentiality) and application data has no protection. For example, a company may wish to protect its code from visual inspection. Since there is no need for integrity checking for this application, no storage for time stamps or IC tag values is needed. The TS input to the AES core, shown in Algorithms 1 and 2, are set to zeroes for this case. In all applications except **Hash**, the operating system code and data use both confidentiality and integrity checking. For the **Hash** application, only confidentiality is used for the OS instructions.

Tables 4 and 2 summarize the task, external memory count, and number of memory segments for the applications. Note that tasks only represent application tasks, not operating system tasks, but memory segments include ap-

Application	Programmable protection									
Application	Total		AES-TASC unit		IC unit		SMM		Control	
	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs
Image	3328	1048	1584	434	319	153	144	31	1281	430
VOD	3311	1042	1586	432	319	153	108	27	1298	430
Comm	3273	1042	1608	439	320	154	41	10	1304	430
Hash	2355	898	1282	434	0	0	8	1	1005	413
	Uniform protection									
Application	Total		AES-TASC unit		IC unit		SMM		Control	
	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs	ALUTs	FFs
Image	3149	1020	1501	433	319	154	15	3	1314	430
VOD	3299	1054	1561	440	407	177	19	3	1312	434
Comm	3141	1022	1503	436	317	153	13	3	1308	430
Hash	2613	904	1518	438	0	0	14	3	1021	413

Table 3: Detailed breakdown of hardware security core (HSC) resource usage

plication and OS external memory segments. All four applications were successfully implemented on a Stratix II Development Kit board containing a 2S60 FPGA device. Area and embedded memory counts were determined following compilation with Altera Quartus II. Power measurements were determined using the experimental setup and using Altera PowerPlay. Core power for the FPGA was provided by a Tektronix power supply. A multimeter configured as an ammeter was used to measure the current. Other board peripherals (flash memory, DDR SDRAM memory) were powered with a standard power supply input.

5 Experimental Results

For comparison purposes, a NIOS II based system without a security core was compiled to a Stratix II 2S60 FPGA. This base configuration includes data and instruction caches, a timer, flash memory controller, DDR SDRAM memory controller and an JTAG interface. After compilation with Quartus II it was determined that the base configuration with 512 byte caches consumes 4909 ALUTs and operates at a clock frequency of 121 MHz. A base configuration with 2 KB caches requires 4 additional ALUTs and operates at the same clock frequency.

As stated in Section 3, the availability of a security core which allows for different security levels for different memory segments provides for security versus resource tradeoffs. In our analysis we consider three specific scenarios:

- *No protection (NP)* - This is the base NIOS II configuration with no memory protection.
- *Programmable protection (PP)* - This NIOS II and security core configuration provides exactly the security required by each application memory segment (Section 4).
- *Uniform protection (UP)* - This NIOS II and security core configuration provides the highest level of security required by a memory segment to all memory segments. Since all segments use the same security level, the SMM is smaller.

The logic overhead of the security core in the programmable protection case is not constant since the size of the SMM depends on the number of defined security areas.

5.1 Area Overhead of Security

As shown in Table 5 for configurations with 512 byte caches, in most cases the hardware security core (HSC) logic required for programmable protection is greater than for uniform protection due to the inclusion of the SMM. The same clock frequency as the base configuration (121 MHz) was achieved for enhanced configurations. Area results for 2 KB caches are nearly identical and have been omitted. A detailed breakdown of the size of individual units in the HSC is provided in Table 3 for both uniform and programmable protection. It is notable that the ALUTs required for IT storage for the programmable protection version of **VOD** is reduced versus uniform protection since the amount of IT storage is reduced. For the **Hash** application, integrity checking is not performed for either uniform or programmable protection so no hardware material is needed for the IC unit.

Archi	Uniform protection				Programmable protection			
	NIOS + HSC		HSC		NIOS + HSC		HSC	
	ALUT	FF	ALUT	FF	ALUT	FF	ALUT	FF
Image	7971	4569	3149	1020	8165	4603	3328	1048
VOD	8159	4602	3299	1054	8157	4592	3311	1042
Comm	7975	4576	3141	1022	8112	4584	3273	1042
Hash	7326	4405	2553	854	7086	4397	2295	848

Table 5: Architectural parameters for different security levels

5.2 Performance Cost of Security

The run time of each NIOS II-based system for each application was determined using counters embedded within the FPGA hardware. Table 6 shows the run time of each application on each configuration and an assessment of performance loss versus the base configuration. Experiments were performed for all three security approaches using both 512 byte and 2 KB caches.

The percentage performance loss due to security is higher for configurations which include smaller caches. This is expected, since smaller caches are likely to have a larger number of memory accesses, increasing the average fetch latency. Some per-application variability is seen. Both **image processing** and **VOD** applications show a substantial performance reduction (23% and 21%, respectively) with uniform protection even though both contain data segment which require no protection. The use of programmable protection allows these data segments to have less of an impact on application performance. Modest performance reductions (14% and 13%) are reported for these configurations.

Note that for the 2 KB cache versions of the **communication** application, the performance loss for the programmable protection version is only 2% less than the uniform protection version. Table 2 shows that all data and code for this application must be protected with either confidentiality or confidentiality and integrity, so the benefit of programmability is limited.

	No protection	Uniform protection		Programmable protection	
	(ms)	(ms)		(ms)	
Image 512	93.3	121.4	-23%	108.7	-14%
Image 2k	65.5	80.4	-18%	73.5	-11%
VOD 512	8186.5	10307.7	-21%	9405.6	-13%
VOD 2k	5369.0	6387.0	-14%	6076.0	-12%
Comm 512	42.8	53.1	-20%	49.0	-13%
Comm 2k	26.4	29.5	-10%	28.8	-8%
Hash 512	5.3	6.4	-18%	6.2	-15%
Hash 2k	3.9	4.5	-15%	4.3	-14%

Table 6: Application execution time and performance reduction

5.3 Memory Cost of Security

As stated in Section 4, memory overhead is the result of on-chip storage of time stamp and integrity tags. Equation 1 provides formulae needed to obtain the amount of on-chip memory which will be needed to store these values. For our experimentation, the IC unit input size is 128 bits, the IC unit output size is 32 bits, the cache line size is 256 bits, and the time stamp size is 32 bits. Using the values from Table 2, it is possible to determine the size of required on-chip memory based on the selected security policy. An example of TS and IC tag overhead calculation is shown for the image processing application with programmable protection.

Figure 4 assesses the on-chip memory overhead of security. Since time stamp and IC tag values consume secured on-chip embedded memory, the availability of embedded memory for other applications is reduced. For the **VOD** application, 150 kB of on-chip memory are saved by using programmable protection rather than uniform protection. The large savings primarily results from the presence of a large unprotected memory segment in the **VOD** application which doesn't require protection. Note that the programmable protection version of the **Hash** application does not require any memory storage since no data values require integrity protection and TS values are not needed for read-only application code.

Equation 1 - ***Security memory equation***

Size of IC memory for code:

$1 - IC\ overhead = \frac{\frac{cache\ line\ size}{IC\ unit\ input\ size} * IC\ unit\ output\ size}{cache\ line\ size}$

$2 - IC\ code = Total\ code * IC\ overhead$

Size of IC memory for data:

$3 - IC\ data = Total\ data * IC\ overhead$

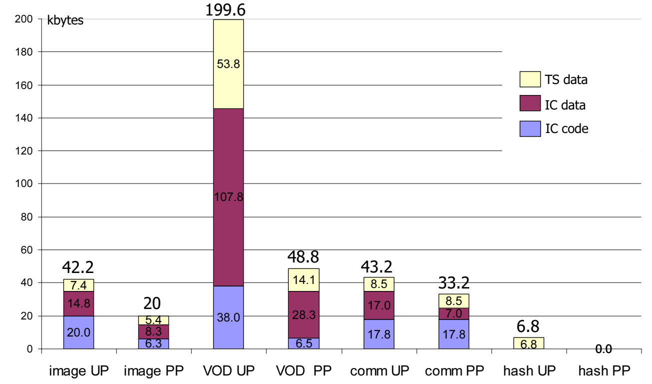

Figure 4: On-chip security memory footprint

Size of TS memory for data:

$4 - TS\ overhead = \frac{TS\ size}{cache\ line\ size}$

$5 - TS\ data = Total\ data * TS\ overhead$

Example for image processing with programmable protection:

$IC\ overhead = \frac{\frac{256}{128} * 32}{256} = 0.25$

$IC\ code = 25kB * 0.25 = 6.25kB$

$IC\ data = 33kB * 0.25 = 8.25kB$

$TS\ overhead = \frac{32}{256} = 0.125$

$TS\ data = (33kB + 10kB) * 0.125 = 5.4kB$

5.4 Energy Cost of Security

Table 7 provides an energy comparison of the four applications, as physically measured in the lab. Energy, instead of power, is used here to take into account the different run times of the applications for different security policies. Cache size has a significant impact on the energy results since larger cache sizes require fewer memory transactions.

Application	No protection	Programmable protection		Uniform protection	
Image 512	84 mJ	112 mJ	+33%	125 mJ	+49%
Image 2k	60 mJ	77 mJ	+27%	84 mJ	+38%
VOD 512	7.02 J	9.7 J	+38%	12.4 J	+76%
VOD 2k	4.9 J	6.3 J	+28%	7.7 J	+57%
Comm 512	34 mJ	53 mJ	+58%	58 mJ	+72%
Comm 2k	22 mJ	31 mJ	+42%	33 mJ	+47%
hash 512	4.7 mJ	6.7 mJ	+42%	7 mJ	+50%
hash 2k	3.5 mJ	4.7 mJ	+33%	4.9 mJ	+40%

Table 7: Energy consumption comparison

Our security core is designed to avoid dynamic power consumption via clock gating when it is not active. The percentage of energy saved by programmable protection versus uniform protection depends on the application. Since **VOD** uses a substantial amount of unprotected memory it exhibits only a 28% energy increase versus the base configuration versus a 57% energy increase for uniform protection with a 512 byte cache. The **communications** application contains no unprotected memory and therefore shows only a few percentage points energy savings between programmable and uniform protection implementations. With larger cache

memories, the overhead is less important. The **image processing** application does not show large savings since the amount of on-chip memory is relatively small. Most of the energy savings comes from a reduced number of HSC accesses.

5.5 Appropriateness for Reconfigurable Systems

Although the security approach described in this paper could possibly be used for a processor-based ASIC, it currently is optimized for FPGA implementation. As stated in Section 4, each application has a custom-sized SMM and time stamp and IC tag storage. If a new application is targeted to the FPGA, the configuration bitstream is changed and a new memory security policy is put in place. The bitstream encryption capabilities of the FPGA allow for this secure change in security policy. Although not covered in this paper, we envision our memory security approach as a first step towards providing a capability to securely download a new FPGA configuration remotely to an embedded FPGA, which then operates in a fashion that ensures application and data security. Each new design is memory-protected with minimal system resource usage, a substantial benefit for many resource-constrained embedded systems.

In contrast, the implementation of our new memory security approach on an ASIC, although possible, would be more difficult and lead to inefficiencies. Such an ASIC would likely be built to support a number of applications, not just one, to minimize overhead. As a result, the SMM and associate time stamp and IC tag storage would have to be sized to fit the *maximum* size required by an application. For example, if the ASIC needed to be able to switch between the four applications described in Section 5, a total memory of at least 48.8 KB must be allocated for TS and IC tag storage. Unlike the FPGA implementation, which only requires a new encrypted bitstream, switching between applications for the ASIC requires added encryption circuitry to prevent unwanted misconfigurations of the SMM and tag storage, a potential burden for the ASIC designer. Additionally, an ASIC implementation precludes the possibility of increasing tag and SMM storage or modifying security steps if new memory protection approaches are discovered after device deployment.

6 Conclusions

In this paper we present a security approach for external memory in FPGA-based embedded systems. Our FPGA-based security core provides both confidentiality and integrity for data stored externally to an FPGA which is accessed by an on-chip processor under operating system control. The benefits of our security core are demonstrated using four embedded applications implemented on a Stratix II device. Average memory and energy savings of about 64% and 16%, respectively, are achieved for the four applications versus a uniform security approach. Overall, the inclusion of security slows application performance by 12% versus execution with unprotected external memory. This value compares favorably to previous approaches which exhibit over a 50% penalty.

Acknowledgments

The authors wish to acknowledge Altera Corporation's donation of the NIOS II Development Board and Quartus II software. Romain Vaslin is partially funded by the Collège Doctoral International of the Université Européenne de Bretagne. Deepak Unnikrishnan is supported by National Science Foundation grant CNS-0708273.

References

[1] Altera Corporation. *Nios II Development Kit*, July 2007.

[2] R. Anderson. *Security Engineering*. John Wiley & Sons, 2001.

[3] T. Eisenbarth, T. Guneysu, C. Paar, A.-R. Sadeghi, D. Schellekens, and M. Wolf. Reconfigurable trusted computing in hardware. In *Proceedings of the ACM Workshop on Scalable Trusted Computing*, 2007.

[4] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, M. Bardouillet, and A. Martinez. A parallelized way to provide data encryption and integrity checking on a processor-memory bus. In *Proceedings of the IEEE/ACM International Design Automation Conference*, July 2006.

[5] J. LaBrosse. *MicroC/OS-II: The Real-Time Kernal*. CMP Books, San Francisco, CA, 2002.

[6] D. Lie, C. Thekkath, and M. Horowitz. Implementing an untrusted operating system on trusted hardware. In *Proceedings of the ACM Symposium on Operating Systems Principles*, October 2003.

[7] A. J. Menezes, P. C. van Oorschot, and S. A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[8] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, 2003.

[9] G. E. Suh, C. W. O'Donnell, I. Sachdev, and S. Devadas. 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*, 2005.

[10] R. Tessier and W. Burleson. Reconfigurable computing and digital signal processing: A survey. *Journal of VLSI Signal Processing*, 2001.

[11] R. Vaslin, G. G. ans Jean-Philippe Diguet, E. Wanderley, R. Tessier, and W. Burleson. Low latency solution for confidentiality and integrity checking in embedded systems with off-chip memory. In *Workshop on Reconfigurable Communication-centric Systems-on-Chip*, June 2007.

[12] R. Vaslin, G. Gogniat, J.-P. Diguet, R. Tessier, and W. Burleson. A security approach for off-chip memory in embedded microprocessor systems. *submitted to Elsevier Journal of Microelectronics and Microprocessors*, 2008.

[13] T. Wollinger, J. Guajardo, and C. Paar. Security on FPGAs: state-of-the-art implementations and attacks. *ACM Transactions on Embedded Computing Systems*, 3(3), 2004.

[14] 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 International Symposium on Computer Architecture*, june 2006.