# High Efficiency Protection Solution for Off-Chip Memory in Embedded Systems

Romain Vaslin, Guy Gogniat, Jean-Philippe Diguet, Russell Tessier, Wayne Burleson

# High-efficiency protection solution for off-chip memory in embedded systems

**Romain Vaslin, Guy Gogniat**
**Jean-Philippe Diguet**
University of Bretagne Sud
LESTER CNRS FRE 3427
Rue de Saint Maud
56321 LORIENT Cedex France
email: vaslin@univ-ubs.fr

**Russell Tessier, Wayne Burleson**
University of Massachusetts
Dept of Electrical and Computer Engineering
309G Knowles Engineering Building
AMHERST, Mass. 01003
email: tessier@ecs.umass.edu

**Abstract** *This paper proposes a complete hardware solution for embedded systems that fully protects off-chip memory. Our security core is based on one-time pad (OTP) encryption and a CRC32 integrity check module. These modules safeguard external memories for embedded processors against a series of well-known attacks, including replay attacks, spoofing attacks and relocation attacks. The implementation limits memory space overhead to about 18.75% and reduces memory latency from 14 cycles for a alternate approach to 3 clock cycles. A FPGA-based implementation of the security core has been completed to gauge the security overhead and to compare our approach with existing solutions.*

*Keywords:* embedded systems, security, cryptography

## 1 Introduction

With the development of new wireless communication standards like WIFI and Bluetooth, inter-entity communication (cell phone, PDA) is becoming unavoidable. Since sensitive data are often exchanged (e.g. a credit card number), it is necessary to protect these transfers. Security is quickly becoming a main bottleneck for communicating entities especially for embedded systems where performance is limited. More and more systems are facing hardware and software attacks [1]. Several solutions have been proposed that protect system architectures (secure architecture) and the data which is transferred (cryptography). Architecture protection mainly corresponds to the protection of data and the program stored in the system memory. Communication protection is related to the protection of data exchanged over an insecure communication channel.
As a consequence, various solutions have emerged that improve system protection. It is essential that these solutions support hardware architectures for embedded systems that meet tight constraints on memory size, performance and power consumption. In the following sections we propose a solution to fully protect an external memory (confidentiality and integrity) of embedded systems.
The paper is organized as follows. Section 2 describes the threat model and state of the art existing solutions. Section 3 details the one-time pad (OTP) protection and necessary extensions for integrity checking. In section 4, a typical implementation of our solution which uses an Altera NIOS II embedded processor [2] is described. Finally, section 5 offers perspectives on this work.

## 2 State of the art

### 2.1 Threat model

As described in [3], the external memory of an embedded system can face a variety of attacks, including those involving probing of the bus between a processor core and the memory. Often, an adversary can easily examine the data and address values placed on a bus. If the bus data is sensitive it must be ciphered with an encryption algorithm such as 3DES [4] or AES [5]. In this case, the confidentiality will be guaranteed. With spoofing attacks, relocation attacks or replay attacks, data ciphering along does not provide a sufficient level of security. A spoofing attack (Figure 1) occurs when an attacker provides a random data value on the bus, causing the system to malfunction. A relocation or splicing attack (Figure 2) occurs when an instruction put on the bus by an attacker is copied from a different bus address. If the whole memory is encrypted with the same key, the swapped instruction will be executed instead of the original instruction. For example, a swapped instruction could make the program jump to malicious code stored in a non-ciphered part of memory. The last type of attack a system might face is a replay attack (Figure 3). This attack is similar to a relocation attack but an attacker provides a data value that was previously located at an address before it was overwritten.

### 2.2 Existing solutions

This section describes three existing memory protection solutions. Two of these approaches, XOM [6] [7] and AEGIS [8] [9] [10], also provide other security primitives such as secure context switching and security level management. However, in this paper we solely evaluate techniques for external memory protection. For

each technique there are system concerns which impact cores in the designated secure area.

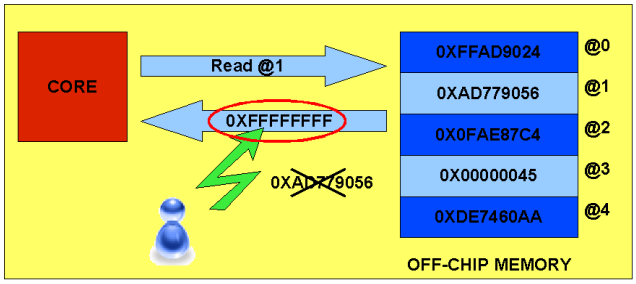


Figure 1: Spoofing attack

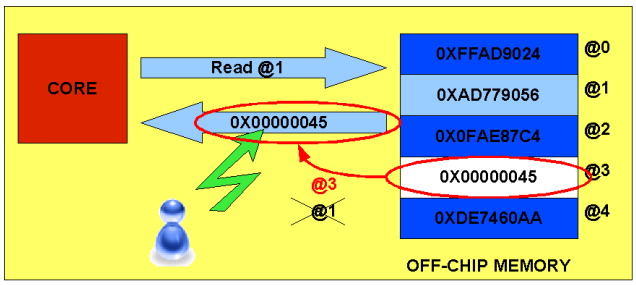


Figure 2: Relocation attack

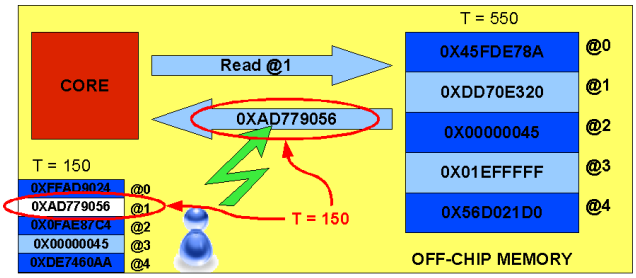


Figure 3: Replay attack

### 2.2.1 XOM

The *eXecute Only Memory* (XOM) [6] [7] approach which provides memory protection, is based on complex key management (Figure 4). Each memory partition is associated with a session key that is needed to decrypt its contents (shown in Figure 4). Encrypted sessions keys are stored in main memory and can be decrypted using an asymmetric secure private key. Decrypted session keys are stored in the XOM key table. The key (private key in Figure 4) required for the asymmetric decryption is stored in the secure zone of the architecture. The algorithm used for the symmetric deciphering is an AES 256. When the core produces a cache miss, the 256 bits read from the memory need to be decrypted. For this case, AES increases the memory latency (case a on Figure 8).
Data integrity is ensured by a message authentication code [11]. A hash of the data and its virtual address is concatenated with the data. The hash is then ciphered

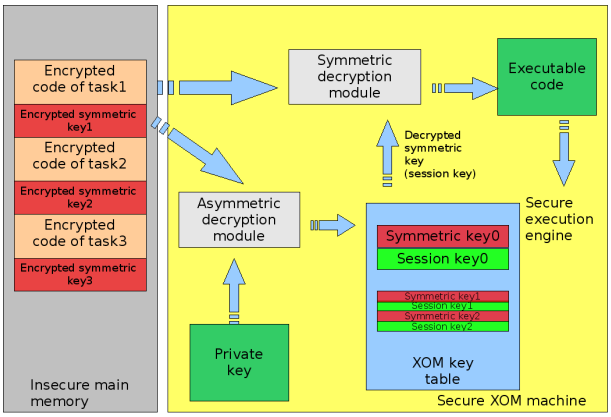


Figure 4: XOM architecture

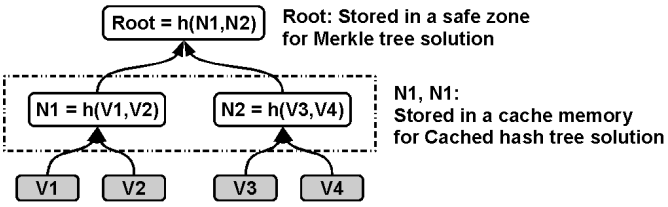


Figure 5: Hash tree

with the data and stored in memory. Although effective, this solution does not protect the system against replay attacks.

### 2.2.2 AEGIS

AEGIS [8] [9] [10] is an additional memory security solution. The confidentiality in the AEGIS solution relies on OTP encryption. The mechanisms used in OTP will be detailed in section 3.1. This encryption method typically has a small impact on memory latency at the cost of memory space overhead.
The solution used by AEGIS for integrity checking is called a *cached hash tree*. This hashing approach is similar to a *Merkle tree* [12] but to increase the efficiency of the method some hash tree nodes are stored in a cache memory (Figure 5). For Merkle trees, only the hash root of tree is securely stored. All hashes must traverse the tree until the root is reached. For cached hash trees, a hash is only performed until the desired node is found in the hash. As a result, cached hash trees offer better results than Merkle trees. Cached hash trees can only be considered secure if the hash cache memory is in a trusted area of the system.

### 2.2.3 PE-ICE

PE-ICE [3] uses the spreading feature of block ciphering algorithms for AES to provide system confidentiality and integrity. Like XOM, a tag is added to the data before ciphering (Figure 6). For read-only values, the tag includes the memory address to prevent relocation attacks. For read-write values, the address and a random value are included to prevent replay attacks. Due

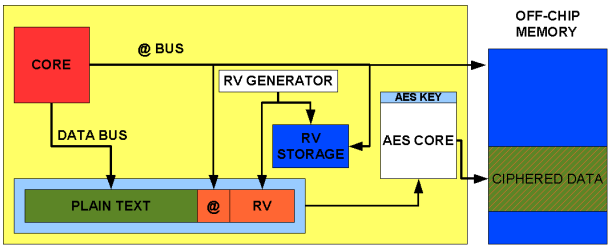


Figure 6: Write request with PE-ICE

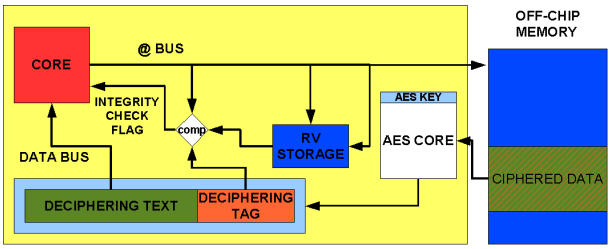


Figure 7: Read request with PE-ICE

to the spreading feature of AES, if one memory bit is modified, a huge impact will appear in the deciphered value. Indeed, the output of an AES is influenced by the input. The plaintext is composed of the data and the tag. When the system performs a comparison between the deciphered tag and the original one concatenated with the data, it can detect if data integrity has been maintained (Figure 7). Like XOM, PE-ICE can have an impact on memory read latencies since decryption can only be performed after the read of a full cache line from external memory. Integrity checking is added just with a comparator for the address and the tag. So the amount of logic needed to guarantee integrity is not important.

# 3 OTP encryption with extensions for integrity checking

## 3.1 OTP encryption standard solution

OTP encryption was initially proposed by Gilbert Vernam during World War I [13], but was only recently adapted for digital memory protection [10]. This previous work proposed to use the memory read access time to compute a random key called an OTP. This key is then XOR'd with the ciphered data to obtain the retrieved plaintext. Each OTP is created before a memory write and is used for encryption. The same OTP is used for subsequent decryption.
In most systems, memory accesses require a long latency. As a result, the cache line read latency may be long enough to perform OTP computation with AES. The AES algorithm is used to generate a random key. As shown in Figures 8(b) and (c), the latency added by encryption is reduced compared to *case a* which represents previous solutions (XOM, PE-ICE). These previous solutions use the data to be stored as the input for AES. In the case (b) on Figure 8, the latency added by OTP encryption is only the latency of a logical XOR operation. In general, the time needed to retrieve the data from the memory for decryption is longer than the time needed to compute the OTP with AES.
From a security standpoint, it is essential that the OTP key is used only one time. The OTP key is obtained with AES, so the AES inputs also need to be used just one time. If an OTP key is used several times, information leakage may occur. The attacker may be able to determine if data ciphered with a same OTP have the same values. In some cases, this leakage could be considered to be a problem depending on the level of desired security.
Since OTP computation is supported by AES, the inputs to AES must be determined. To prevent a system against relocation attacks, the data memory address is used as an AES core input for OTP generation (Figure 9). To prevent replay attacks, time stamps (TS) are used. As shown in Algorithm 1, the TS value associated with each data address is incremented by 1 after each OTP generation. For each new cache line memory write request, the system will compute a different OTP since the value of TS is incremented. The TS values are stored in a memory for later use during memory read operations. 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 result of AES will give the same OTP as the one produced for the write request and the encrypted data will become plaintext after being XOR'd (Algorithm 2). Read-only data does not require protection against replay attacks because these data are never modified. No TS values are needed for these data so the amount of TS memory space can be reduced. Read-only data may be the target of relocation attacks but the address used to compute the OTP guarantees protection against these attacks. The size of the address and the TS might not be long enough to completely fill the AES function input, so padding may be necessary. A random value (RV) is used to pad the input value.

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

$1 - CRC\,(@) = CRC\{plaintext\}$
$2 - Time\ stamp\ incrementation:\ TS\,(@) = TS\,(@) + 1$
$3 - OTP\ computation:\ OTP = AES\{TS\,(@), @, RV\}$
$4 - Ciphered\ data = plaintext \oplus OTP$
$5 - Ciphered\ data \Rightarrow memory$
$6 - TS\,(@) \Rightarrow TS\ memory$
$7 - CRC\,(@) \Rightarrow CRC\ memory$

The use of time stamp and data addresses for OTP protects a system against replay and relocation attacks. If data is replayed, the TS used for ciphering will differ from the one used for deciphering. If data is relocated, its address will differ from the one used to generate the OTP. In both cases, the deciphered data will be invalid. To use this information, the secure memory access system must be able to detect that the deciphered data is incorrect. Thus, we present an extension to the OTP encryption in the next section.

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

$1 - Get\ TS\,(@) \Leftarrow TS\ memory$
$2 - Get\ CRC\,(@) \Leftarrow CRC\ memory$
$3 - OTP\ computation:\ OTP = AES\{TS\,(@), @, RV\}$
$4 - Get\ ciphered\ data \Leftarrow memory$
$5 - Plaintext = Ciphered\ data \oplus OTP$
$6 - CRC\,(@) \equiv CRC\{plaintext\}$
$7 - Plaintext \Rightarrow cache\ memory$

*Highlighted operations are only available for the extended OTP solution proposed here with integrity checking*

Our OTP implementation is efficient because it performs OTP computation (operation 3 in Algorithm 2) in parallel with memory data requests (operation 4 in Algorithm 2). The Figure 8 provides a view of the gain.

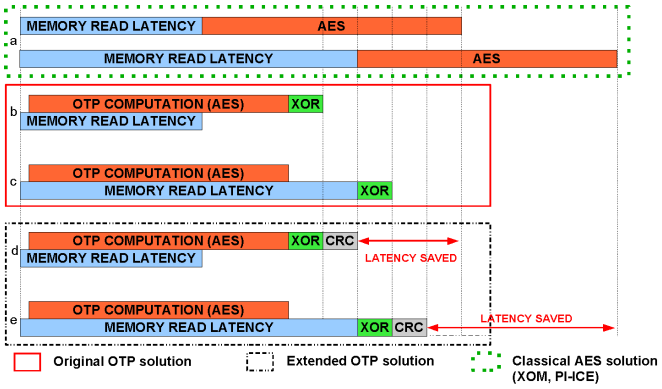


Figure 8: Overview of the latency added by different security solutions for different memory read latencies. Cases a-top, b and d: the memory read latency is shorter than AES computation. Cases a-bottom, c and e: the memory read latency is longer than the AES computation.

Our OTP implementation is useful because it performs OTP computation (operation 2 in Algorithm 2) in parallel with memory data requests (operation 3 in Algorithm 3). The Figure 8 provides a view of the gain.

## 3.2 Integrity checking extension

The system must be able to produce an error if an OTP core indicates an OTP mismatch. Therefore, a detection mechanism is needed. Additionally, integrity checking must be performed with a negligible overhead to minimize latency. Our solution to this issue involves the use of a CRC32 module. Prior to OTP generation, the CRC32 of the cache line (operation 1 in Algorithm 1) to be encrypted is stored in a cache (operation 7 in Algorithm 1). Later, when the processor core requests a read, the CRC32 result of the final XOR operation is compared with the CRC32 value stored in the memory (operation 6 in Algorithm 2). If data is changed following storage, the CRC32 of the retrieved value will differ from the stored value, so the attack is detected. As previously stated, the results of decryption following a replay or relocation attack will differ, so the CRC will differ. As shown in Figure 8 the latency added to the original OTP solution by our extension is the latency of CRC computation and checking. This CRC computation can be completed in one clock cycle. With the extended OTP, the minimum latency added to a memory access is the time to obtain the result of the XOR and the CRC check (Figure 8).

# 4 Implementation with an embedded processor

## 4.1 Global architecture features

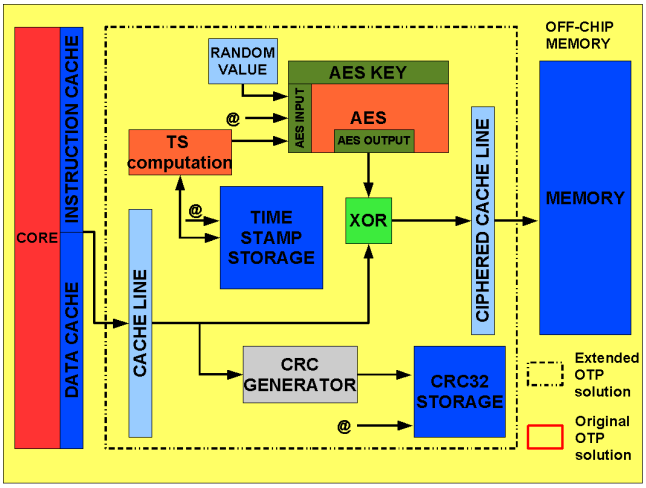


Figure 9: OTP write request

The Altera NIOS II embedded processor has been used to test our new memory protection approach. The chosen configuration includes both an instruction and a data cache, each with 512 bytes and a 256 bits cache line. As seen in Figures 9 and 10, NIOS caches are interconnected to the OTP design via a 32 bits connection. A 32-bit wide connection is also used to connect to 4 Mbits of SDRAM.
For this work, we assume that the OTP core cannot be attacked using techniques such as fault injection. The memory space required to store the time stamps and CRC32 values depends on the nature of the stored data. Overheads are summarized in Equation 1.
As an example, we consider a system with a total memory size of 512 KB. A total of 256 KB is read-only data and the remaining 256 KB is read-write (RW) data. According to Equation 1 we need to have $OTP_{\text{STORAGE}} = 96$ KB (32 KB for $TS_{\text{STORAGE}}$ and 64 KB for $CRC32_{\text{STORAGE}}$ with a $TS\ SIZE$ and a $CRC32\ SIZE$ of 32 bits ). Time stamps are unnecessary for read-only data.

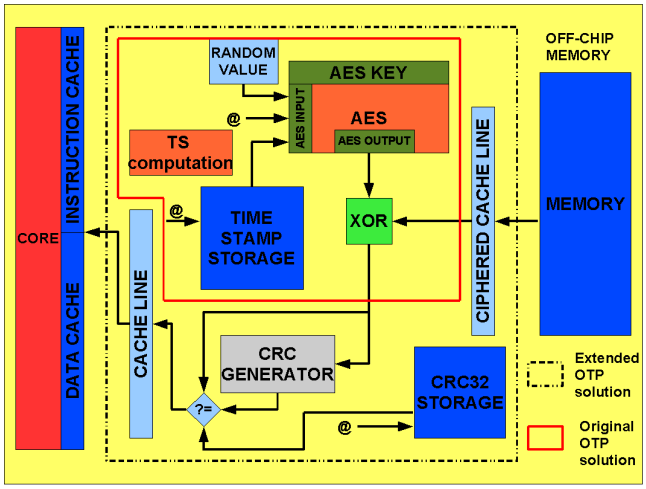


Figure 10: OTP read request

**Equation 1 - *OTP memory consumption***

$$OTP_{\text{STORAGE}} = TS_{\text{STORAGE}} + CRC32_{\text{STORAGE}}$$

$$TS_{\text{STORAGE}} = \left(\frac{RW\ DATA\ MEMORY\ SIZE}{CACHE\ LINE\ WIDTH}\right) * TS\ SIZE$$

$$CRC32_{\text{STORAGE}} = \left(\frac{TOTAL\ MEMORY\ SIZE}{CACHE\ LINE\ WIDTH}\right) * CRC32\ SIZE$$

For our example system, an AES core of 128 bits is selected to minimize the hardware impact of OTP on the overall design. As a result, OTP values are 128 bits long. As described in section 3.1, each OTP value should not be used more than once. Since the AES core generates 128 bits and each cache line has 256 bits per line, each 128 bits OTP must be used twice to encrypt a full cache line. In this case, information leakage is not a significant enough concern to warrant a 256 bits OTP. The adversary may be able to determine that the first 128 bits of the OTP are the same as the 128 last bits (Equation 1). If a more secure implementation is required, a 256 bits AES implementation may be used (Equation 2). In Section 4.2, we evaluate the cost of both the AES 128 and AES 256 solutions. The CRC32 module has an input of 256 bits (a full cache line). This module produces a 32 bits output which is stored in the CRC32 cache (Figure 9) or compared with a value stored in the cache (Figure 10).

**Equation 2 - *Two security levels***

$$1 - OTP_{128} = AES_{128}\{TS(@), @, RV\}$$
$$1 - Ciphered\ data_{256} = plaintext_{256} \oplus \{OTP_{128}, OTP_{128}\}$$
$$2 - OTP_{256} = AES_{256}\{TS(@), @, RV\}$$
$$2 - Ciphered\ data_{256} = plaintext_{256} \oplus OTP_{256}$$

| | Base NIOS | NIOS + OTP128 + CRC | | NIOS + OTP256 + CRC | |
|---|---|---|---|---|---|
| | | | overhead | | overhead |
| Logic (ALUTs) | 2134 | 6193 | x2.90 | 6767 | x3.17 |
| Memory (KB) | 512 | 600.56 | +18.75% | 603.12 | 19.7% |
| Read latency (cycles) | 10 | 21 | +11 | 21 | +11 |
| Write latency (cycles) | 0 | 12 | +12 | 12 | +12 |

Table 1: Cost of security for NIOS II

| | OTP128 core + CRC | OTP control | AES128 | CRC32 |
|---|---|---|---|---|
| Logic (ALUTs) | 4059 | 1918 | 1479 | 662 |
| Memory (KB) | 98.56 | 32 | 2.56 | 64 |
| | OTP256 core + CRC | OTP control | AES256 | CRC32 |
| Logic (ALUTs) | 4633 | 1999 | 1972 | 655 |
| Memory (KB) | 101.12 | 32 | 5.12 | 64 |

Table 2: OTP core overhead breakdown

| | total latency | AES (11) | data (8) fetch | XOR (2/1) | CRC32 (1) |
|---|---|---|---|---|---|
| Read latency (cycles) | 11 | - | x | x | x |
| Write latency (cycles) | 12 | x | - | x | - |

Table 3: OTP core latency

## 4.2 Cost of security

In this section, we present the cost of adding our memory protection mechanisms to a NIOS II based system. In Table 1, it can be seen that the impact on the design logic size in look-up tables (ALUTs) is significant (x2.81). Memory overhead is 18.75% for our chosen parameters. As discussed in the previous section, these overheads depend on the memory architecture and desired security level of the system (Equation 1).
The added circuitry has an effect on latency; 11 additional cycles are needed to perform read transactions compare with a base NIOS architecture (security latency on Figure 11). These 11 cycles include 8 cycles to perform the read of a full cache line with OTP. The last 3 clock cycles represent the time needed to perform XOR and CRC check operations on the data (note Table 2). This overhead is significant but as shown in Figure 8 the overhead is less important for our new approach versus previous approaches based on AES protection because the data fetching is done in parallel (Figure 11). With a standard encryption solution, the latency would be the time needed for the data fetching (8 cycles) and the delay of AES computation (14 cycles). It means 22 cycles compared to the 11 of our proposition. For a read request, the base NIOS architecture has an intrinsic latency of 10 cycles, while write latency is null (Figure 11). In the case of our secure architecture, the overhead for a write request is 12 cycles (the security latency on Figure 11). A total of 11 cycles are due to the time required to perform AES.

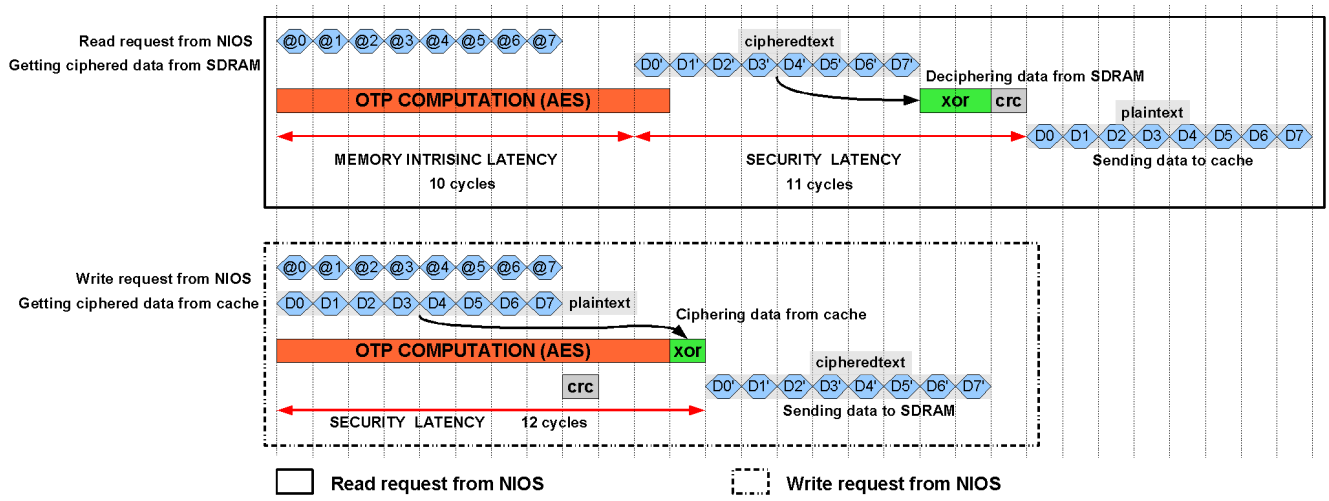


Figure 11: Management of NIOS request with a SDRAM

| | base AES (no integrity) | our solution OTP + CRC | | XOM AES + MAC | | PE-ICE AES | | AEGIS OTP + cached hash trees | |
|---|---|---|---|---|---|---|---|---|---|
| | | | overhead | | overhead | | overhead | | overhead |
| Memory (KB) | 512 | 600.56 | +18.75% | N/A | N/A | 776 | +51.5% | 768 | +50% |
| Read latency(cycles) | 22(14+8) | 11(3+8) | -11 | 22 | 0 | 25(17+8) | +3 | $\approx$ 4715/80 | +4502/69 |
| Write latency(cycles) | 22(14+8) | 12(8+4) | -10 | 22 | 0 | 26(18+8) | +4 | N/A | N/A |

Table 4: Overhead comparison of all the solutions with a classic AES protected solution. The latencies presented, are those added by encryption (the time to fetch data is included, 8 cycles are required)

The last cycle is due to OTP management and control. These latencies are significant but all solutions requiring block ciphering will require some latency. The benefit of a solution depends on the time required to perform encryption (Figure 8). In our case, this time is 1 or 3 cycles but for previous AES solutions (PE-ICE or XOM), about 14 cycles are required.
It is interesting to note that the latency of the $OTP_{128}$ is the same as $OTP_{256}$. A standard AES 256 would require 14 cycles, but for this experiment we implemented two AES 128 cores in parallel. Thus, the latency of the system is the same but the hardware cost is more important.
Tables 2 summarizes the overhead of the OTP cores. In both cases the logic overhead due to the CRC is constant, in contrast to AES. In the $OTP_{128}$ core, the AES 128 represents 36% of the design size and 46% for the 256 bits version. The memory requirement for the AES 256 differs slightly from AES 128 (See Eq. 4.1 for details).

## 4.3 Evaluation

In the previous section, we described the cost of security for our solution. In this section, we compare this cost to previous solutions described in section 2.2. Table 4 summarizes a number of relevant cost values. All of these approaches support the same level of security (confidentiality and integrity for an off-chip memory).
The first desired point of comparison is logic area overhead. Unfortunately, a lack of data from the other approaches made this comparison impossible. In general, each approach requires at least one AES core. Differences include the number of cores used by each solution and the method chosen for integrity check. For PE-ICE, there is no hardware cost for integrity checking. For AEGIS, the integrity check (cached hashed tree) uses an SHA-1 algorithm which is generally performed in software. The software approach can be time consuming. In [9], the authors propose a hardware implementation of SHA-1 although no overhead values are presented. For our case, the logic overhead added by integrity checking is only in the CRC check module.
In terms of memory, our solution consumes less space than other solutions. AEGIS also guarantees confidentiality using OTP so it also requires space for time stamps. However, the use of a cached hash tree for integrity checking causes a memory overhead of 33%. For XOM no memory overhead figures have been published. However, since the XOM integrity check uses a MAC solution some storage space will be needed to store hash signatures. Memory overhead for PE-ICE results from tags (address and random values) added to the data (Figure 6) and also from on-chip storage needed to securely store random values.
A final comparison point is system memory latency. If we compare the latency of our new approach with an earlier AES based solution (such as XOM or PE-ICE), it will be less. For PE-ICE, latency is an issue due to the time needed to check if a tag is the same as one stored in on-chip memory. For AEGIS, which is based on OTP encryption, the latency caused by confidentiality is reasonable, but the integrity check is done in software.

This issue badly impacts the system. For example, in [9], the authors report that the SHA-1 algorithm needs 4715 cycles to compute the hash. If the implementation was done in hardware, the latency would be around 80 cycles which is still significant. It is clearly shown in Table 4 that our approach reduces latency compared to other approaches. Only 3 cycles are needed instead of the 14 cycles required by previous AES based solutions.

# 5 Perspectives

In this paper, we have evaluated the impact of off-chip memory security on a processor architecture and we have verified that the cost of our solution is moderate. A number of interesting issues remain. So far, we have only analyzed the latency to obtain plaintext. The next step is to study the overhead of our solution on software execution. Since many embedded systems require battery-based operation, power consumption is also an important issue. A complete analysis of the power costs of our approach is needed. From a security standpoint, additional work is needed to protect on-chip memory used to store TS and CRC32 values. This memory could be targeted by fault injection attacks leading to incorrect system operation. TS and CRC32 values possibly could be stored in off-chip memory
The work presented in this paper uses a reconfigurable target (FPGA). The features of reconfigurable architectures provide some interesting perspectives for security. It may be possible to adapt the security level of the architecture in response to different threat levels. In [14], the authors propose reconfigurable mechanisms to provide for a fault tolerant AES. Another security adaptation opportunity might involve real-time operating systems (RTOS). The RTOS may have specific primitives to enable hardware security primitives. The isolation of non-sensitive data would reduce the amount of memory needed to store TS and CRC32 tags.

# 6 Conclusion

This paper presents an efficient security solution (confidentiality and integrity) for off-chip memory. OTP encryption is combined with CRC32 integrity checking to reduce memory access latency and secure memory overheads. The demanding requirements of embedded systems have led us to propose a solution for such systems. The next step for this approach might be the implementation of memory overheads in non-secure parts of the architecture which could be exposed to fault injection.

# References


[1] David Dagon, Tom Martin, and Thad Starner. Mobile phones as computing devices: The viruses are coming! *IEEE Pervasive Computing*, 3(4):11–15, 2004.

[2] http://www.altera.com/. ALTERA website.

[3] Reouven Elbaz, Lionel Torres, Gilles Sassatelli, Pierre Guillemin, Michel Bardouillet, and Albert Martinez. A parallelized way to provide data encryption and integrity checking on a processor-memory bus. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 506–509, July 2006.

[4] 3DES RFC 1851. ftp://ftp.rfc-editor.org/in-notes/rfc1851.txt, September 1995.

[5] AES RFC 3565. ftp://ftp.rfc-editor.org/in-notes/rfc3565.txt, July 2003.

[6] David Lie, Chandramohan Thekkath, and Mark Horowitz. Implementing an untrusted operating system on trusted hardware. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 178–192, October 2003.

[7] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural support for copy and tamper resistant software. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 168–177, 2000.

[8] G. Edward Suh, Charles W. O'Donnell, Ishan Sachdev, and Srinivas Devadas. Design and implementation of the aegis single-chip secure processor using physical random functions. In *ISCA '05: Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 25–36, 2005.

[9] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Aegis: architecture for tamper-evident and tamper-resistant processing. In *ICS '03: Proceedings of the 17th annual international conference on Supercomputing*, pages 160–171, 2003.

[10] G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. Efficient memory integrity verification and encryption for secure processors. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 339, 2003.

[11] H. Krawczyk, M. Bellare, and R.Canetti. Hmac: Keyed-hashing for message authentification, February 1997.

[12] R. C. Merkle. Protocols for public key cryptography. In *IEEE Symposium on Security and Privacy*, pages 122–134, 1980.

[13] Ross J. Anderson. *Security Engineering: A Guide to Building Dependable Distributed Systems*. John Wiley & Sons, Inc., 2001.

[14] Wayne Burleson, Guy Gogniat, and Tilman Wolf. Reconfigurable security support for embedded systems. In *HICSS '06: Proceedings of the 39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, page 250.1, January 2006.