FEDTIC: A Security Design for Embedded Systems with Insecure External Memory

Mei Hong and Hui Guo

School of Computer Science and Engineering,
The University of New South Wales,
Sydney, NSW 2052, Australia
meihong@cse.unsw.edu.au, huig@cse.unsw.edu.au

Abstract. This paper presents a security design for embedded systems that have a secure on-chip computing environment and an insecure off-chip memory. The design protects the confidentiality and integrity of data at a low cost on performance and memory consumption. We implemented the design based on the SimpleScalar simulation software. Our simulation on a set of benchmarks shows that very little overhead is incurred for on-chip memory, and the average overheads on performance and off-chip memory, are only 7.6% and 6.25%, respectively.

1 Introduction

Security becomes increasingly important in embedded systems. One driving force is the security-aware services that embedded systems provide, such as financial transactions, application downloading on mobile devices. Embedded software systems are vulnerable, as they are written in insecure lower-level language with poor support for runtime error checking [1]. The embedded systems can also easily be attacked through physical accesses. Therefore, the confidentiality and integrity of the data that are processed and communicated must be ensured.

Processors and memory are two typical components in the embedded system for computing and data storage. With the fast expanding of embedded system applications, the requirement for memory continues to grow. Apart from the memory equipped on the processor chip, large off-chip memory is usually indispensable. However, the link between the off-chip memory and on-chip processor is often a weak security point and the off-chip memory is more vulnerable to variety of attacks.

The conventional way for confidentiality protection is to use encryption techniques. The data exposed to possible attacks are encrypted; Only in a secure environment, can the plain data be viewed and used. When the data are stored in or transmitted to outside of the secure environment, they are always in an encrypted format.

For the integrity protection, a typical solution is tagging the data – attaching a piece of data item (or called **tag**) that is exclusive to the data to be protected. Any alteration to the data by an intruder will result in a corrupted tag, which forewarns the data should not be used.

However, realizing these security protections is often at the cost of performance, hardware area, and energy consumption. In this paper, we propose an efficient, low

T.-h. Kim et al. (Eds.): FGIT 2010, LNCS 6485, pp. 365–375, 2010.

cost security design – a Fused Encryption/Decryption and Tagging/Integrity Checking (**FEDTIC**) engine – for systems that have a secure on-chip computing environment and insecure off-chip memory. We reduce the area overhead by sharing one hardware component for both encryption and decryption. We improve performance by combining the encryption/decryption operation and tagging/integrity checking into one step, and this step is performed in parallel with the memory access so that the impact of the security implementation on the system performance is minimized.

The paper is organized as follows. Section 2 reviews the related work. The *FEDTIC* design is explained in Section 3. The experimental setup and simulation results are presented in Section 4, and the conclusive remarks are given in Section 5.

2 Related Work

There are many types of possible attacks. An overview of them can be found in [2].

The "secure processor" with an insecure external memory was first proposed by Best [3]. According to Best, only the processor chip is secure, other external components are vulnerable to attacks. To increase the security, contents on external memory are encrypted and will be decrypted when they are fetched to the processor. The cipher unit and secret key used for encryption and decryption are kept on-chip. A set of commercial secure processors, Dallas Semiconductor DS5000 series [4], are designed based on this idea and have been used for different applications, such as the pay-TV access controller and credit card terminal.

Blum, etc. [5] proposed an approach for the integrity protection of memory data. Significant efforts have been made to enhance the security for both the confidentiality and integrity of the external memory. Confidentiality is achieved through instruction and data encryption. Integrity verification is accomplished by creating an authentication tag for each memory block using MAC (Message Authentication Code) function or cryptographic hash function. The encryption and integrity verification can be implemented in software or hardware.

Since software-only methods, such as code obfuscation and watermarking, do not resistant physical attacks, intensive researches are devoted on architectural support for secure environment. Two most-referenced approaches are XOM [6,7], and AEGIS [8,9,10]. Both adopt the Best's architecture: secure processor chip and insecure external memory.

XOM provides an architectural support for copy and tamper resistant software, where the execute-only memory(XOM) is implemented that allows instructions stored in external memory to be executed but not otherwise manipulated. Instructions and data are stored in an encrypted form. Integrity is achieved by isolating independent software applications running on the same processor. Each application is stored in a compartment for the secure execution; accessing contents in one compartment by applications from others is forbidden. Tags are created to identify contents in each application and different session keys are used to encrypt associated data.

AEGIS provides both a tamper-resistant environment where attackers are unable to obtain any information from system operation (confidentiality), and a tamper-evident, authenticated environment in which any physical or software tempering is guaranteed to be detected (integrity).

One of the problems in these architectures is the performance overhead. Every external memory transaction including both instructions and data undergoes encryption and decryption. They cannot be used until they are fetched from the external memory and decrypted. Cryptography is a computational intense operation, which greatly degrades performance. Moreover, adding tags for integrity verification results in more memory consumed.

Authors in [11] proposed a "CryptoPage" architecture which implements memory encryption, memory integrity checking and information leakage protection with a low performance penalty. The authors combined the AES encryption in the counter mode of operation with Merkle tree authentication for these encryption, integrity protection security features. The Merkle tree technique decreases the on-chip memory overhead, which is incurred by storing hash values or nonces. Elbaz et al. [12] improved the tree structure by parallelizing the hash tree update operation.

The above methods combine the fast hardware implementation for encryption and Merkle tree for integrity. This basic composition [13] generally uses Encrypt-then-Mac, a two pass approach. Therefore, the cost to achieve confidentiality-and-integrity is the cost of encryption plus the cost of MAC computation. Several one-pass approaches, such as AE modes [14] have been proposed. However, to the best of our knowledge, no implementation of such a design has been reported for external memory protection in embedded systems.

In this paper, we develop a security engine for external memory encryption and integrity verification. This engine can provide the same security level as other existing one-pass designs, but has significant low overheads on both performance and memory consumption.

3 Design Approach

Figure 1 shows the top-level architectural setting of the Fused Encryption/Decryption and Tagging/Integrity Checking (*FEDTIC*) engine in a system with an on-chip processor and an off-chip memory.

The *FEDTIC* is implemented on the processor chip (secure area). For confidentiality, data stored in the off-chip memory are all in an encrypted form (as marked inside the "[]" when they are transferred over the memory bus outside of the processor chip). The tag of data is affixed at the end in the cache line for each transfer between the on-chip cache and off-chip memory. The *FEDTIC* engine provides encryption/decryption, tagging and integrity verification for each of the external memory accesses.

We introduce a value, called Line-Access Stamp (**LAS**) that serves as a hallmark for each cache-line memory access. For every memory write, a new LAS is generated and used to encrypt and tag the cache line that is written to the memory. When read from memory, the memory data are decrypted and the related tag is calculated. The calculated tag is then compared with the tag coming along from the memory. If they are same, the integrity of the memory data is verified and the data is loaded to the cache; otherwise, the memory read operation failed.

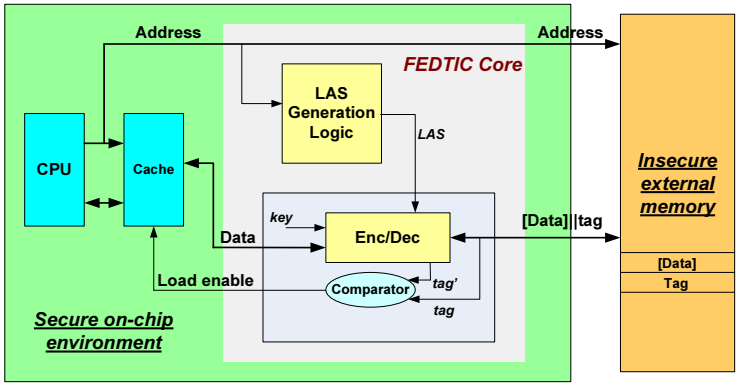

Fig. 1. System Architecture

Different from other designs, our *FEDTIC* engine features following specialities:

- The encryption and decryption operations are identical. Therefore, we use a same hardware component for both encryption and decryption, hence saving the hardware cost.
- We exploit the encryption/decryption process and let the integrity tag generation/checking mix with the encryption/decryption to speed up *FEDTIC* operation; and
- The *FEDTIC* operation is in parallel with the memory access, hence its impact on the overall system performance is reduced to a lowest possible level.

The key parts of the $FEDTIC$ design are detailed in the following subsections.

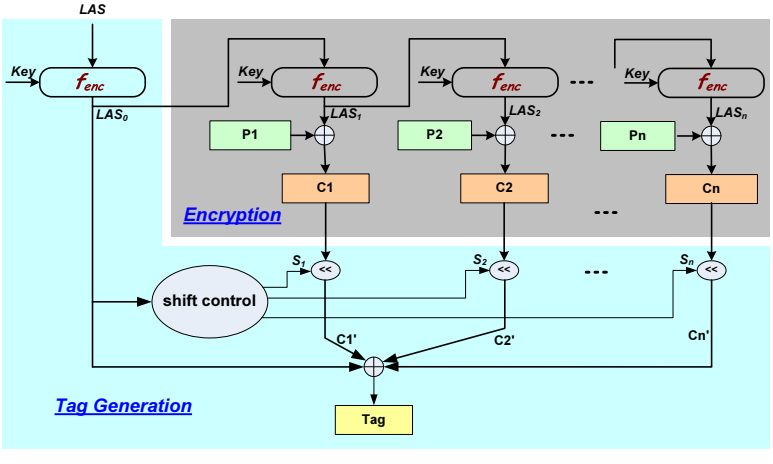

Fig. 2. Encryption and Tag Generation

3.1 Encryption and Tagging

Since the symmetric encryption can run as much as 1000 times faster than the asymmetric encryption, we employ the symmetric encryption in our design. With our design, the encryption and decryption are always performed on the processor chip, hence no need for the encryption key distribution (which would otherwise be an issue, commonly faced by many symmetric encryption applications). We use the symmetric cipher on blocks with fixed length (e.g. 64 or 128 bits). A cache line (data transferred between the cache and external memory for one memory access) consists of multiple such blocks. Therefore, we use block cipher to encrypt a cache line in the mode operation (with the Output FeedBack mode (**OFB**) as showed in Figure 2).

With the OFB mode, instead of encrypting plaintext blocks directly, LAS is recursively encrypted with a symmetrical encryption function, f_{enc}. The output of the previous encryption is the input of the current encryption. Each round of encryption produces a separate LAS value, which is XORed with the plaintext block (denoted as $P1, P2, \cdots, Pn$, respectively) to generate a ciphertext ($C1, C2, \cdots, Cn$, correspondingly), as shown in the region marked as *Encryption* in Figure 2.

Algorithm 1. Encryption process and tag generation

```
/* recursive LAS encryption */
LAS_0 = f_enc(LAS);
for i = 1 to n do
    LAS_i = f_enc(LAS_{i-1});
end for
/* plaintext block encryption */
for i = 1 to n do
    C_i = LAS_i ⊕ P_i;
end for
/* encrypted data: concatenation of all C_i */
C = Null;
for i = 1 to n do
    C = C || C_i;
end for
/* shift control operation for each block based on the bit segments of LAS_0 */
for i = 1 to n do
    m_bit_segment = bit m * (i - 1) to bit m * i - 1 of LAS_0;
    S_i = number_of_zeros in (m_bit_segment);
end for
/* transformed ciphertext blocks: left shift S_i bits for block i */
for i = 1 to n do
    C'_i = C_i << S_i;
end for
/* Tag: XOR of transformed blocks and LAS_0 */
tag = LAS_0;
for i = 1 to n do
    tag = tag ⊕ C'_i;
end for
```

To obtain a cache line tag, the cheap way (often used in traditional designs) is XORing all data blocks in the cache line. The data blocks can be in either a plaintext format or a ciphertext format. But either way invites potential attacks. If the plaintext was used, the original data could likely be deduced due to the easy availability of the tag provided by the insecure memory; On the other hand, if the ciphertext was used, altering data by switching different blocks in the off-chip memory would not change the tag value, hence an integrity attack would easily be applied.

We use a *transformed ciphertext blocks* (by bit shifting operations) in the tag generation. Before the XOR operation, each block is left-shifted and the number of bits to be shifted is controlled by the *Shift Control Logic* which is, in turn, determined by the encrypted *LAS* value, LAS_0, as illustrated in the second region in Figure 2. With this tag design, any swapping of the ciphertext blocks in the external memory will have a different tag value, hence the attack can be identified.

The operations in the encryption and tagging process are summarized in Algorithm 1, where we assume there are n blocks in a cache line and each block has m bits.

From Figure 2, we can see that the encryption process and tag generation can be partially parallelled. After the first round of *LAS* encryption, the shift control logic can perform in parallel with the encryption. All S_i values can be generated once LAS_0 is available; C_i can be calculated when LAS_i is completed; C_i can be transformed (for the tag calculation) as soon as S_i is known. The extra time taken by the tag generation is just the sum of the execution times for one shift and one XOR operation. Therefore, the total execution time is reduced. An parallel execution example is given in Figure 3, where a cache line of 4 blocks is assumed.

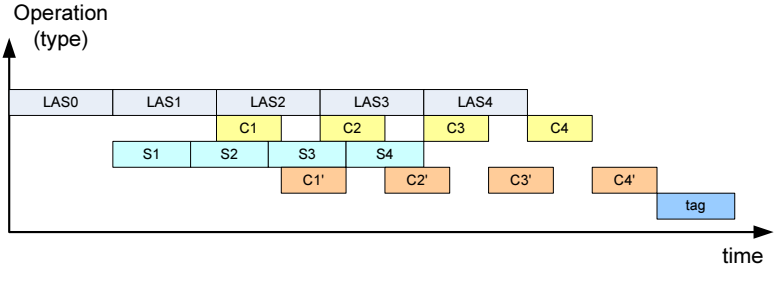

Fig. 3. Parallel Operations in Encryption and Tag Generation During Memory Write

3.2 Decryption and Integrity Checking

The decryption and integrity checking for a cache line that is fetched from memory is given in Figure 4.

It consists of two parts: decryption and tag calculation that is based on the ciphertext blocks from the memory. Because the *OFB* mode of operation is used, as for the same cache line, the same *LAS* is encrypted. The decryption is identical to the encryption process. Because of the symmetrical attribute of the XOR operation, the results of the encryption are XORed with ciphertext to get the plaintext. The tag is calculated in the same way as in the tag generation.

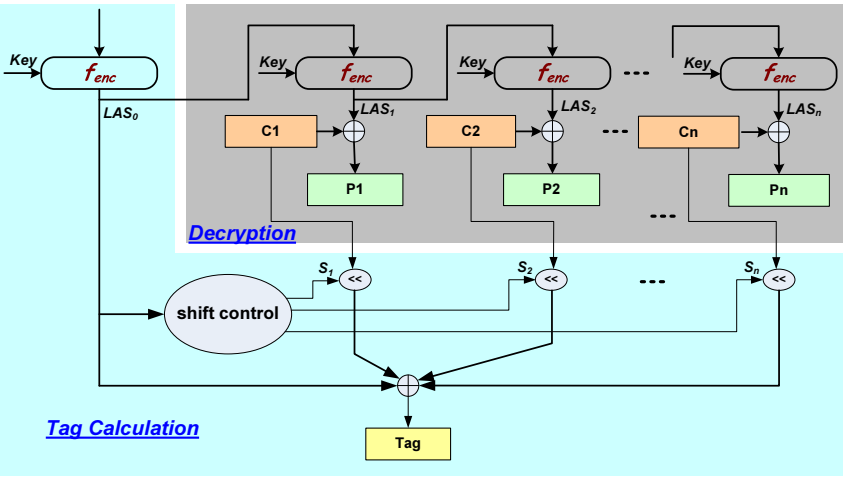

Fig. 4. Decryption and Tag Calculation

Unlike for the memory write operation, where the input plaintext is immediatelyavailable from the cache for encryption, a long latency time is often needed to obtain the ciphertext for a memory read operation. Therefore, the parallel execution is different from that in the memory write operation, as illustrated in Figure 5. As can be seen from Figure 5, the memory latency can be used to perform the time-costly encryption function, f_{enc}; therefore, only the XOR function added to the delay to the critical path for the cache read operation.

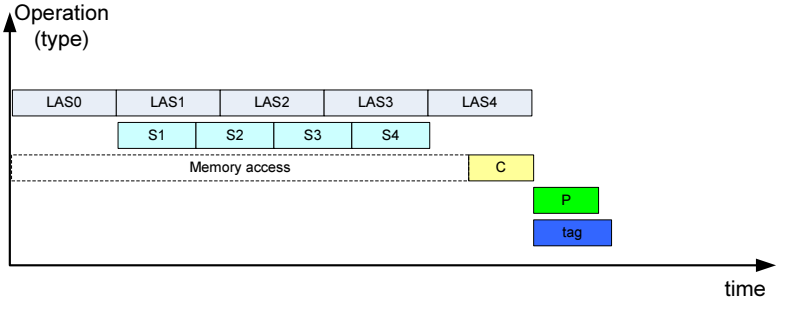

Fig. 5. Parallel Operations in Decryption and Tag Calculation During Memory Read

4 Experimental Results

To evaluate our design, we have built a simulation environment based on the SimpleScalar simulation suite[15]. We use the speculative out-of-order simulator with PISA instruction set architecture. A memory hierarchy that contains two-level instruction and data caches is applied in the architecture.

The SimpleScalar is modified to implement on-chip *FEDTIC* functions. The baseline design architectural parameters used in the simulation are shown in Figure 6.

Architectural Parameters	Specification
Clock Frequency	400MHz
L1 I-caches	64KB, 2-way, 32B line
L1 D-caches	64KB, 2-way, 32B line
L2 I-caches	256KB/1MB/4MB, 2-way, 32B/64B/128B
L2 D-caches	256KB/1MB/4MB, 2-way, 32B/64B/128B
L1 Latency	2 cycles
L2 Latency	10 cycles
Memory Latency (first/following chunk)	18/2 cycles
Memory Bus	200 MHz, 8-B wide (1.6GB/s)
AES Latency	20 cycles
Counter	8 bits
Random Number	32 bits

Fig. 6. Architectural Parameters

We chose AES as our encryption function. The Xilinx FPGA implementation of AES presented in [16] was integrated in our design. Seven SPEC2000 [17] CPU benchmarks and one MiBench[18] benchmark ($stringsearch$) were used in the simulation.

In our simulation parameters, the processor speed is 400MHz and the *FEDTIC* operates on a FPGA at 200MHz. Therefore, every FPGA computation cycle is equivalent to two processor cycles. The external bus and off-chip memory is assumed to run at 200MHz.

We evaluate the performance overhead from the security design compared to the baseline system without the security protection engine.

The overhead is incurred when a cache miss occurs. On a read miss, the requested cache line is fetched from the external memory, decrypted and tag-checked; on a write miss, the cache line to be written to the external memory is encrypted and tagged. Thus, the overall performance penalty is affected by the cache miss rate, and cache miss penalty. Different cache configurations (cache size, cache line size) affect cache miss rate. We design our experiments with different cache configurations in order to observe the performance overheads. We add a fixed penalty for each memory fetch.

Figure 7(a) shows the baseline performance for each of the benchmarks with different cache sizes (256KB, 1MB, and 4MB) and the same 64Byte cache line. The performance is measured in Instruction Per Cycle (IPC).

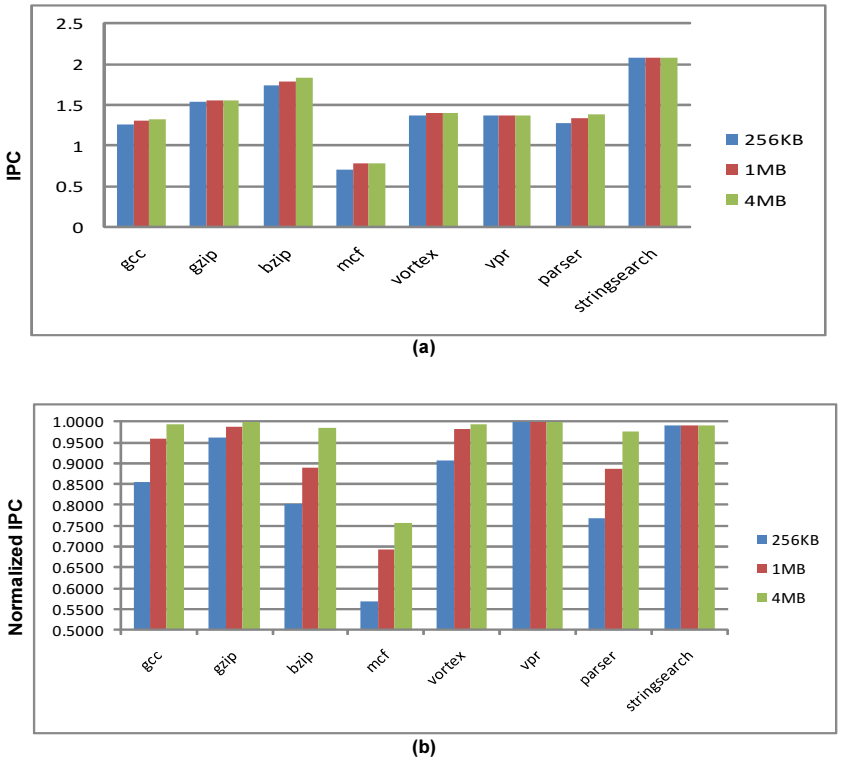

Fig. 7. Effect of Different Cache Configuration (a) Baseline Performance (b) Performance Overheads

Figure 7(b) illustrates the impact of security protection on the run-time program performance, where the IPC is normalized to the baseline for an easy comparison.

From Figure 7(a), we can see that with increasing cache sizes, the performance is slightly improved, by 2.10% on average. Figure 7(b) shows that the security protection results in some performance overhead that is inversely related to the cache size within the range of 256KB-4MB. With the 256KB-cache, the overhead is up to 43.14% (the worst case from the mcf application) and averaged around 14.36%. When the cache size is increased to 1MB, the average overhead is reduced to 7.66%, and the worst case is improved, to 30.72%, and for some benchmarks, the overhead is less than 5%. When the cache size is further increased to 4MB, the average overhead is only 3.84%, and the worst case is 24.29%.

Table 1 shows the memory and performance overheads for each of the applications when the system has a $1MB$ cache with the cache line of $64Bytes$. The on-chip and off-chip memory costs for each of the benchmarks are given in columns 2 & 4, the percentage values as compared to the baseline design are listed in columns 3 & 5. The IPC values for the baseline design and the design with *FEDTIC* are presented in columns 6 & 7, and the performance overhead as compared to the baseline design is given in the last column (column 8).

Table 1. Memory and Performance Overheads

benchmarks	memory overhead				performance overhead		
	on-chip	%	off-chip	%	baseline (IPC)	FEDTIC (IPC)	%
gcc	4KB	0.39%	173KB	6.24%	1.3130	1.2577	4.21%
gzip	5KB	0.49%	37KB	6.28%	1.5529	1.5348	1.17%
bzip	1KB	0.10%	15KB	6.28%	1.7880	1.5923	10.95%
mcf	0.3KB	0.05%	9KB	6.43%	0.7805	0.5407	30.72%
vortex	2KB	0.20%	69KB	6.23%	1.3954	1.3684	1.93%
vpr	0.6KB	0.06%	25KB	6.25%	1.3645	1.3643	0.01%
parser	4KB	0.39%	30KB	6.25%	1.3451	1.1938	11.25%
stringsearch	0.4KB	0.04%	7KB	6.54%	2.0746	2.0525	1.07%
average		0.21%		6.25%			7.6%

From the simulation results, we can see that our security design incurs little overheads, on average, about 6.25% on the off-chip memory, 7.6% on the system performance, and only 0.21% on the on-chip memory, for the design with a $1MB$ cache and the cache line of $64Bytes$.

5 Conclusion

In this paper, we presented an efficient encryption/authentication scheme to protect the confidentiality and integrity of the data that are processed in a system which has a secure on-chip computing environment and insecure off-chip memory. Our design is easy to implement. We have modeled our design based on the Simplescalar simulation software. Our experiment on a set of benchmarks demonstrates that our security design incurs very little on-chip memory consumption, about 0.21%, and the overheads on the off-chip memory and the system performance are only 6.25% and 7.6%, respectively.

References

1. Gelbart, O., Leontie, E., Narahari, B., Simha, R.: A compiler-hardware approach to software protection for embedded systems. Computers and Electrical Engineering, 315–328 (2009)
2. Ravi, S., Raghunathan, A., Chakradhar, S.: Tamper resistance mechanisms for secure embedded systems. In: 17th International Conference on VLSI Design (2004)
3. Best, R.M.: Prevent software piracy with crypto-microprocessors. In: IEEE Computer Society International Conference (1980)
4. Dallas Semiconductor (2008), `http://www.maximic.com/Microcontroller.cfm`
5. Blum, M., Evans, W., Gemmell, P., Kannan, S., Naor, M.: Checking the correctness of memories. In: 32nd Annual Symposium on Foundations of Computer Science (1991)
6. Lie, D., Chandramohan, T., Mitchell, M., Lincoln, P., Boneh, D., Mitchell, J., Horowitz, M.: Architectural support for copy and tamper resistant software. In: 9th Internatinal Conference Architectural Support for Programming Languages and Operating Systems, ASPLOS-IX (2000)

7. Lie, D., Thekkath, C.A., Horowitz, M.: Implementing an untrusted operating system on trusted hardware. In: 19th ACM Symposium on Operating System Principles (2003)
8. Suh, G.E., Clarke, D., Gasend, B., van Dijk, M., Devadas, S.: AEGIS:architecure for tamper-evident and tamper-resistant processing. In: International Conference on SuperComputing (2003)
9. Suh, G.E., óDonnell, C.W., Sachdev, I., Devadas, S.: Design and implementation of the AEGIS single-chip secure processor using physical random functions. In: 32nd Interntional Symposium on Computer Architecture, ISCA (2005)
10. Suh, G.E., óDonnell, C.W., Sachdev, I., Devadas, S.: AEGIS: A single-chip secure processor. IEEE Design and Test of Computers, 467–477 (2007)
11. Duc, G., Keryell, R.: Cryptopage: An efficient secure architecture with memory encryption, integrity and information leakage protection. In: 22nd Annual Computer Security Applications Conference, ACSAC 2006 (2006)
12. Elbaz, R., Champagne, D., Lee, R.B., Torres, L.: Tec-tree: a low-cost, parallelizable tree for efficient defense against memory replay attacks. In: Paillier, P., Verbauwhede, I. (eds.) CHES 2007. LNCS, vol. 4727, pp. 289–302. Springer, Heidelberg (2007)
13. Bellare, M., Namprempre, C.: Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. Journal of Cryptology 21(4), 469–491 (2008)
14. Rogaway, P., Bellare, M., Black, J., Krovetz, T.: OCB: a block-cipher mode of operation for efficient authenticated encryption. In: ACM conference on Computer and communications Security (2001)
15. Austin, T.M., Burger, D.B.: The simplescalar tool set, version 3.0. Technical report, University of Wisconsin-madision (1997)
16. Helion Technology Datasheet: high performance AES (Rijndael) cores for Xilinx FPGAs (2008), `http://www.heliontech.com`
17. Henning, J.L.: SPEC CPU 2000: Measuing CPU performance in the new millennium. IEEE Computers (2000)
18. Guthaus, M.R., Ringenberg, J.S.: Mibench: a free, commercially representiative embedded benchmark suite. In: IEEE 4th Annual Workshop on Workload Characterization (2001)