IBC-EI: An Instruction Based Compression method with Encryption and Integrity Checking

Eduardo Wanderley[1,3], Reouven Elbaz[2, 4], Lionel Torres[2], Gilles Sassatelli[2], Romain Vaslin[3], Guy Gogniat[3] and Jean-Philippe Diguet[3]

[1]Dept. Informatics, CEFET-RN, wanderley@cefetrn.br
[2]LIRMM Laboratory, CNRS, Université de Montpellier II CNRS, [torres, sassatelli]@lirmm.fr
[3]LESTER Laboratory University of South Britany [vaslin, gogniat, diguet]@univ-ubs.fr
[4]Dept of Electrical Engineering, Princeton University, relbaz@princeton.edu

Abstract

Code Compression has been shown to be efficient in minimizing the memory requirements for embedded systems as well as in power consumption reduction and performance improvement. In this paper we devise a code compression method, the IBC-EI (Instruction Based Compression with Encryption and Integrity checking), tailored to provide integrity checking and encryption to secure processor-memory transaction. The principle is to keep the ciphered code compressed in the memory, thus reducing the memory footprint and providing more information per memory access. The results for the Leon Processor and a set of Mediabench and MiBench benchmarks show that the overhead introduced by the code encryption and integrity checking scheme is almost completely eliminated by the compression mechanism.

Keywords: Code Compression; Compression; Encryption; Integrity Checking, Security;

1 Introduction

Embedded systems have been used in a large myriad of applications and nowadays the crescent demand for services from remote and/or mobile terminal are coming more and more important. Many of these services involve private information, like bank account numbers, thus making security an important issue today.

In this scenario, we found the processor-memory information traffic as a weakness in security. Observing memory contents flow, to and from the processor, reverse engineering of the software and/or access to private or sensitive data can be accomplished [1,2]. Those board level attacks challenge data confidentiality as well as data integrity.

In order to ensure confidentiality of processor-memory transactions, encryption of the memory footprint of the embedded software is implemented. On the other hand, data integrity is ensured by using meta-data, called TAG. Those TAG are computed with MAC – Message Authentication Code – functions [16] or are nonce (redundancy) added to plaintext block before block encryption [6]. The main shortcoming of integrity checking techniques is the off-chip memory consumption for tag storage [3, 4, 5, 6].

Code Compression (CC) was first idealized to respond to the memory constraints of embedded systems. When first introduced [7], a decompressor engine was located between the cache and the main memory of a system, thus the decompression overhead could be hidden by the cache (a block decompression occurs only on cache misses). This scheme imposes a penalty in execution time due to decompression. On the other hand less accesses to the main memory is expected, which, in turn, overcomes the performance problem.

In this paper we propose the Instruction Based Compression with Encryption and Integrity checking – IBC-EI. IBC-EI is the combination of a decompression core with the Parallelized Encryption and Integrity Checking Engine (PE-ICE [5, 6]) which is based on a single block encryption to provide data confidentiality and integrity. The objective of this combination is to decrease the off-chip memory overhead and the performance hit produced by the decryption and integrity checking processes of PE-ICE. In this work we focus on the protection of code (i.e. Read Only data), thus the encryption and the compression of application code can be done off-line. However decryption and decompression take place during software execution thus for performance reason, those process are performed in hardware. This paper describes how works this hardware component at run-time.

This paper is organized as follows: in Section 2 the threat model of the system is described and the existing techniques ensuring code confidentiality and integrity are described. Section 3 presents the proposed code compression method. Section 4 shows the results for the Leon Processor [8] and a set of benchmarks from MiBench [9] and Mediabench [10] suites; in the Section 5 a discussion is presented, inclusive with other state of the art possibilities; finally, we present our conclusions and future work in Section 6.

2 Threat Model and Existing Countermeasures

2.1.Threat Model

The threat model used in this work considers the SoC as trusted. The attacks to be avoided are aimed at the bus activity and the off chip memory. They are called board level attacks and consist of an adversary that probes the bus between the external memory and the SoC. The goals are to retrieve (and possibly understand) the information transmitted (passive attacks challenging data confidentiality) and/or being able to interfere in the execution trace (active attacks challenging data integrity) with or without the knowledge of the consequences the change will produce.

We are particularly interested in spoofing attacks in which an adversary changes aleatoric bits in the memory or on the bus and disturbs the program behavior randomly. Splicing attacks are also addressed. They consist in moving a block of memory to another address. This block of memory, having a known behavior, can help the attacker on finding its contents and its relation with the remaining of the code. Such an attack can be seen as a spatial permutation of memory block in memory. Note that previous works also dealing with board level attacks considers replay attacks (temporal permutation of memory block); however this attack do not apply to our threat model since we only consider code (Read Only data are not sensitive to replay since they are written in memory once and are not modified at run-time).

2.2.Existing Techniques for Encryption and Integrity Checking

In order to prevent the attacks previously listed we must provide data integrity and confidentiality.

To ensure data integrity, Message Authentication Code – MAC – algorithms can be used [11]. The purpose of such algorithms is to give a compact representative image – called in the following TAG or fingerprint – of the message at their input and of its source. To do so the MAC function is applied on the original plaintext by enrolling a secret key.

For the confidentiality aspect an encryption algorithm is used. There are two main families of such algorithms: stream cipher and block cipher. In the first one the encryption is done bit per bit while in the second the message is split into blocks and then each block is encrypted separately.

The conventional way to provide both integrity checking and confidentiality is to use one of three different schemes: Encrypt-then-MAC, MAC-then-Encrypt and MAC-and-Encrypt. Figure 1 shows these three possibilities, from top to bottom respectively. Encrypt-then-MAC is the most implemented scheme since proved secure [12]. However, in all cases during the ciphering, or during the deciphering, or both, the MAC and Encryption/Decryption units are not parallelizable; it ensures an impact on performance and/or on silicon area to implement any of these schemes.

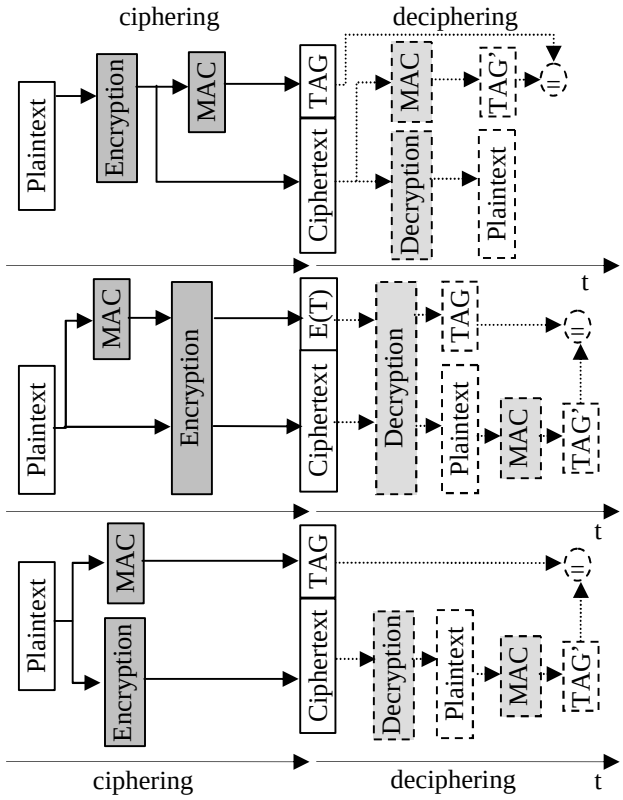

Fig. 1. Confidentiality and integrity integration schemes.

Thus, in this paper, we will use the concept of block-level AREA (Added Redundancy Explicit Authentication) introduced in [5, 6]. This principle leverages the diffusion property of block encryption to add the integrity checking capability to this type of encryption algorithm. This is achieved by applying the AREA technique at the block level [13]: redundant data (e.g. a nonce – a Number used ONCE) is added to each plaintext block before encryption and checked in the decrypted ciphertext block. Upon a memory write, the SoC appends an *n*-bit nonce to the data to be written to memory, encrypts the resulting plaintext block and then writes the ciphertext to memory.

The encryption is performed using a key securely stored on the SoC. The SoC decrypts the block it fetches from memory on a read transaction and verifies that the last n bits of the resulting plaintext block are equal to the nonce that was inserted by the SoC upon encryption. Since a single block encryption invocation is required to provide both data confidentiality and integrity, the concept of block-level AREA is parallelizable on read and write operations and is efficient in term of silicon area required (one block encryption algorithm implementation instead of a encryption and a MAC algorithms in conventional approaches).

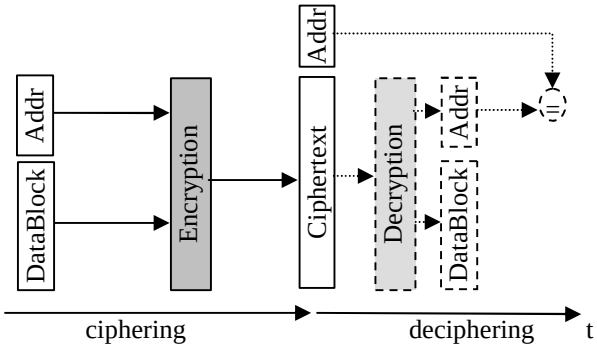

Fig. 2. Parallelized confidentiality and integrity checking scheme.

As mentioned above, we focus on RO data which are not modified during software execution Therefore we use the address of each plaintext block as redundancy since it is a nonce: the data and its address are concatenated, encrypted and stored in the memory. Whenever the processor accesses the block, using its address, the ciphertext block is decrypted and the address that comes up will be checked against the address of the block provided by the processor. If they match the block is considered valid. The first implementation of the block-level AREA concept, called PE-ICE was proposed in [5] and is shown in Figure 2. The overhead of PE-ICE when compared to the AES (Advanced Encryption Standard [14]) encryption is negligible in term of *hardware* area and low in term of run-time performance [6].

. Nevertheless, this approach will still incur performance penalties and, more inconvenient, a considerable off-chip memory overhead (between 25 and 50%). The off-chip memory area used to keep the same program will be increased by the TAG of the block (its address).

In this work we propose to compress the code to compensate the memory overhead generated by the TAG required by the integrity checking process. Moreover, code compression means that more information is brought on-chip with a single memory access and thus improving run-time performance.

3 IBC-EI: the Instruction Based Compression with Encryption and Integrity Checking

Code compression is aimed at minimizing the constrained memory area requirement for embedded systems. As a side-effect, the code compressed in memory is transferred to the SoC using less memory accesses (or more information is transferred per memory access), thus improving performance and reducing energy consumption. Unfortunately, the decompressor used to restore the original information will introduce some penalty in performance. However, this penalty can be, and it is usually the case, hidden by the expensive memory operations reduction.

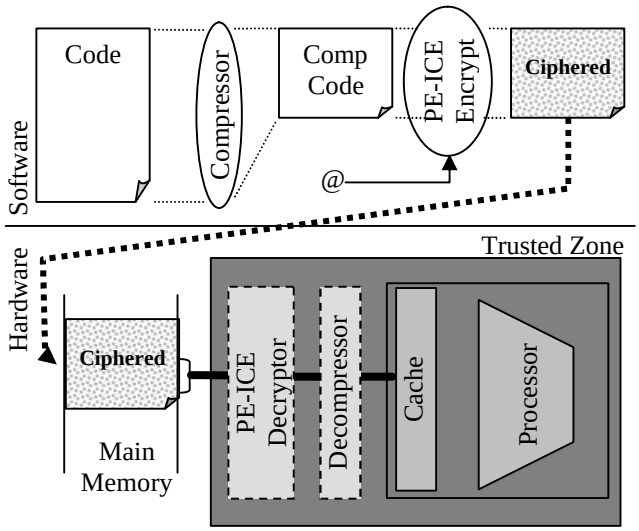

Fig. 3. Code Compression integration with security

The idea to add code compression to the (en|de)cryption and integrity checking scheme is based on the fact that the main degradation (performance hit and memory overhead) introduced by the underlying cryptographic functions can be solved by compression.

The framework for the proposed integration method can be seen in Figure 3. First the original code is compressed and then encrypted. This phase is executed offline, so that the compression algorithm performance is not critical, as well as the encryption process. Then, the ciphered code is stored in the main memory. During the code execution, a block of memory is retrieved and delivered to the processor. The Decryption Unit deciphers, checks the block and transfers it to the Decompression Unit, which, in turn, decompresses the block and delivers the instructions to the Cache/Processor.

The Decryptor and the Decompressor are allocated into the trusted zone (i.e. SoC). The Decompressor will be utilized only on cache misses, thus reducing the possible negative impact on performance.

To provide integrity checking, instead of using an expensive hash algorithm, the block-level AREA technique and PE-ICE model (AES based) are used.

3.1.Advanced Encryption Standard

The Advanced Encryption Standard is a symmetric-key encryption algorithm adopted by the National Institute of Standards and Technology [14].

AES uses a block of 128 bits of data to be encrypted with key lengths of 128, 192 or 256 bits. The AES implementation consists of 10 to 14 rounds of four transformations, depending on the key length chosen. Thus, the encryption (and the decryption associated) will impose an important overhead in the system performance.

3.2.Tailoring the Compression Method

The compression method described here is based on a dictionary approach in which the instructions in the original code are substituted by an index into the dictionary [15]. The index size, IDX_{size}, is a function of the number of dictionary entries, $DICT_{entries}$, as shown in Equation 1.

$$IDX_{size} = \lceil \log_2 DICT_{entries} \rceil \quad (1)$$

The more instructions are in the dictionary, the greatest are the indexes. The compressed code will be a sequence of indexes and its size will be determined by the number of original instructions in the code multiplied by the index size and possibly some final padding bits to fulfill a memory word (Equation 2). Moreover, the dictionary size, in bits, will be the instruction word size multiplied by the number of dictionary entries (Equation 3).

$$COMP_{size} = \#InstOrignalCode \times IDX_{size} + Padding \quad (2)$$

$$DICT_{size} = DICT_{entries} \times InstructionWord_{size} \quad (3)$$

The metric used to evaluate a code compression method will be the Compression Ratio, which represents the size of the compressed code (and the dictionary associated) over the original code size, as defined in Equation 4. This metric, sometimes presented in percentage, has the lowest value as the best one.

$$\text{Compression Ratio} = \frac{COMP_{size} + DICT_{size}}{ORIGINAL_{size}} \quad (4)$$

This simple approach is still insufficient because the code flow can be break by a branch instruction. In this case the processor is supposed to begin decompressing the next instruction from an address and offset different of the following. Possible solutions are aligning every code target and patching the branch instructions offsets or using an Address Translation Table, ATT, mapping original addresses to compressed code addresses and offsets. The ATT size, which is impractical, can be reduced if not every address is mapped, but only a block of addresses. This is intuitive for the case of decompression between the cache and the main memory, as far as the cache line will be the block of information to be fulfilled by the decompressor. Thus, the ATT will keep just the compressed addresses corresponding to the original cache line addresses (and the offset associated). The ATT can also be implemented as a linear function since the indexes have fixed sizes.

Some stronger compressions can be obtained by using multiples dictionaries of varying sizes [15]. In this approach the smaller dictionary, which will have the smaller indexes associated, will keep the instructions that appear the most in the original code.

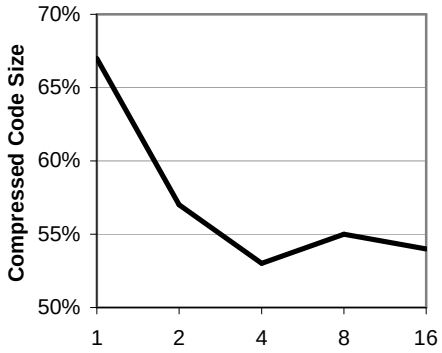

Fig. 4. Compressed Code size for multiples dictionaries usage relative to the original

On the other hand, to identify the dictionary, to which the index in the compressed code belongs, a prefix is necessary. A typical curve for the compressed code size as a function of the number of dictionaries used can be seen in Figure 4. Normally, using 4 dictionaries with 2 bits prefix will lead to better compression of the code. Then, we will use a set of four dictionaries for our experiments.

3.3.Integrating Confidentiality and Integrity Checking

In our description we use the term *chunk* to indicate the atomic block loaded from memory for decryption and integrity checking. A chunk is composed of the block address (the TAG used to check data integrity) and one or more compressed cache line – also called the payload of a chunk. A compressed cache line is the representation of every original cache line with prefixes (*p*) and indexes (*i*) into the dictionaries. This chunk is encrypted with the AES and stored in the main memory. To match the AES pattern a chunk will be, necessarily, 128bit long. If a compressed cache line extrapolates the chunk threshold – or more accurately the payload threshold – it will be delocalized and inserted in the next chunk. This will avoid the use of two AES decryptions to fulfill a cache line whenever necessary, although the compression results will be negatively affected.

The ATT will map the address of every original cache line to the address of the chunk in which the compressed line is present plus the offset, in bits, of where the corresponding line starts.

Figure 5 shows a conceptual view of a typical chunk in memory before encryption. Such a chunk contains two compressed cache lines. In this case, the original cache line size is 16 bytes or 4 instructions. Supposing that the next compressed cache line does not fit the remaining bits of the chunk it will be assembled in the next one and padding is inserted at end of the current chunk.

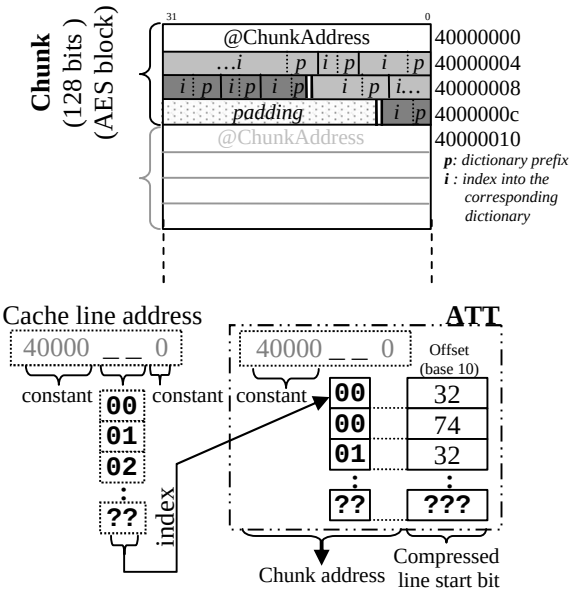

Fig. 5. Typical chunk pattern before encryption used in the IBC-EI composed by two compressed cache lines and the correspondig ATT.

In the figure we can also observe the creation of the ATT. For every cache line in the original code an address mapping is present. Actually, only the original addresses bits used to indicate the cache line serve as indexes into the ATT. Moreover, some bits of the addresses are constant (and program size dependents), so that, only the varying bits are used and stored, reducing considerably the ATT size. The offset is always 7 bits long to be able to represent any of the 128 bits possible starting offsets in the chunk. The first cache line (40000000_h) is mapped into the position of the corresponding compressed chunk, for example, 40000000_h. The offset of this first line is 32. The second cache line (40000010_h) is mapped into the same chunk address but the offset is now 74. The next cache line (40000020_h) will be mapped into the next chunk (40000010_h), as only two compressed cache lines are present in the first chunk.

The main components of the decompressor engine, shown in Figure 6, are the ATT, the set of Dictionaries, and the AES deciphering unit. Whenever a cache miss occurs, the cache line address is presented to the decompressor engine. The first address goes through the ATT and the converted chunk address is found. A copy of this address is kept in one of the inputs of the Comparator (Comp). Then, the encrypted chunk is transferred to the Decompressor Engine. The AES deciphering unit (AES^{-1}) is used to retrieve the original compressed block that is kept in a buffer. Once the decryption is done, the portion of the information used to hold the TAG is compared with the first address of the block to check for integrity. If they match, the block is valid and the decompression begins. A series of shifts obtains the prefixes and indexes into the dictionary. Finally, the instructions can be delivered to the processor. If a second line is requested by the CPU the ATT converts the address and the comparator is used to check if the cache line is already buffered in the Engine. If so, no memory access is required, neither the AES^{-1} unit usage. To provide the correct sequence of addresses to the main memory, an adder is used to increment the initial position supplied by the ATT.

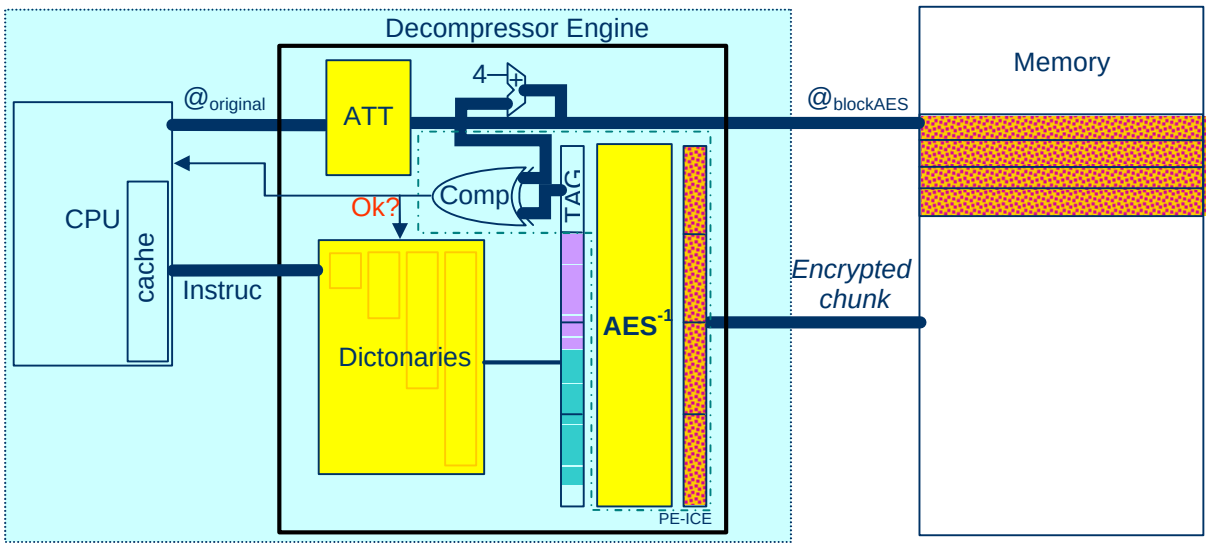

Fig. 6. The decompression engine with security issues integrated

A FPGA implementation of the compression method without the cryptographic functions implemented, revels that it imposes a four-cycle penalty for the decompression itself [15]. The present version of the decompression core will be not very different in term of hardware requirement as the logic is the same, with just the comparator in addition but with easier fetch-from-memory control logic. Note that the expensive deciphering operation required by PE-ICE will be used over compressed data, which means that more information will be brought on-chip on each memory access (and thus saving block decryption invocation). This means that decryption is only required upon decompressor necessity, not necessarily on every cache miss.

Table 1. Simulator Architectural Parameters.

Parameter	Specification
I-Cache	1k, 8k / 16B lines, Direct Mapped
D-Cache	8k / 16B lines, Direct Mapped
Memory latency	80 cycle to retrieve a cache line
AES latency	17 cycles
Decompression latency	4 cycles

4 Results

The processor targeted is Leon (SPARC V8) open source processor [8]. The benchmarks are extracted from MiBench [9] and Mediabench [10]. They are a string search algorithm, Search, commonly used in office suites; Dijkstra, an algorithm used in network routers; Djpeg and Cjpeg, used for compressing and decompressing images from and to JPEG formats; and Adpcm that encodes or decodes audio.

We used LECCS, a GCC based compiler for the Leon processor, with –O2 option in all the benchmarks, so that we avoid typical optimizations that increase object code size, like function in-lining and loop unrolling.

The general setup for the simulator is shown in Table 1. Moreover, to highlight the advantage of IBC-EI we evaluate PE-ICE and AES encryption in the same simulation framework.

We begin by showing the compression ratio obtained in comparison to the original code, including each component of the new compressed code: the ATT; the Dictionaries; and the Compressed Code itself with the associated TAGs. Figure 7 shows the Compression Ratio obtained for each benchmark. On average, an overhead of only 1% in relation to the original code size is obtained. A maximum of 19% is obtained for the djpeg. When PE-ICE is implemented in addition to code compression, the increase of 50% in the code memory area, caused by the TAG usage for each cache line, is completely overcome by compression. This result comes from the fact that the compression reduces the size of the code, thus requiring less TAGs to ensure the integrity of the whole memory footprint and on the other hand frees memory space for the remaining TAGs.

Notice that the off-chip memory will keep only the compressed code. The ATT and the Dictionaries will be kept on-chip.

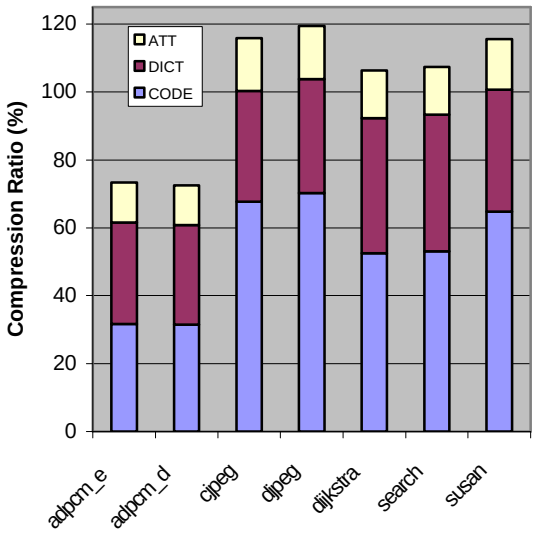

Fig . 7. Compression Ratio of IBC-EI

The compressed code components present the ATT contribution with 14% of the compressed footprint; while the contribution from the dictionaries is 35%; and the compressed code contribute with 51%, on average. These values depend on the instructions redundancy found in the original code, which allows the compression. The higher is the redundancy, the smaller are the dictionaries. Thus, the smaller is the compressed code.

The next experiments show the performance of the system in relation to the one without cryptographic and compression functions implemented – called in the following "base execution" since used as reference. In order to provide a minimal comparison we also show the case in which only confidentiality is addressed with AES encryption. The PE-ICE-only proposal is also estimated. The experiments use two i-cache sizes, as stated in Table 1.

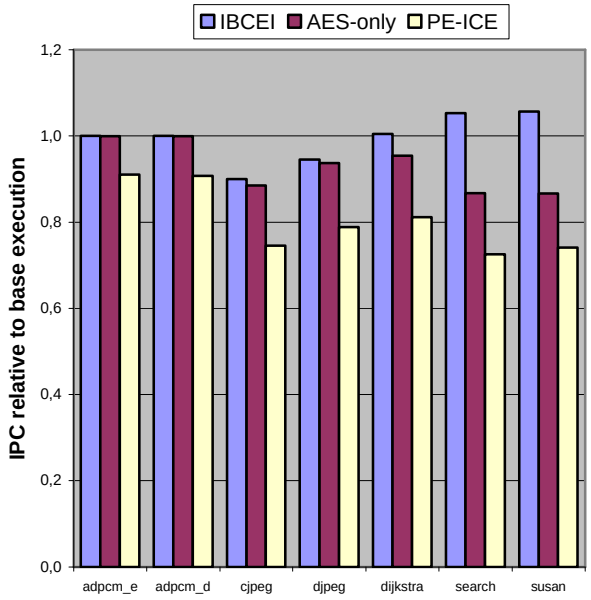

Fig. 8. Performance for the 1k i-cache

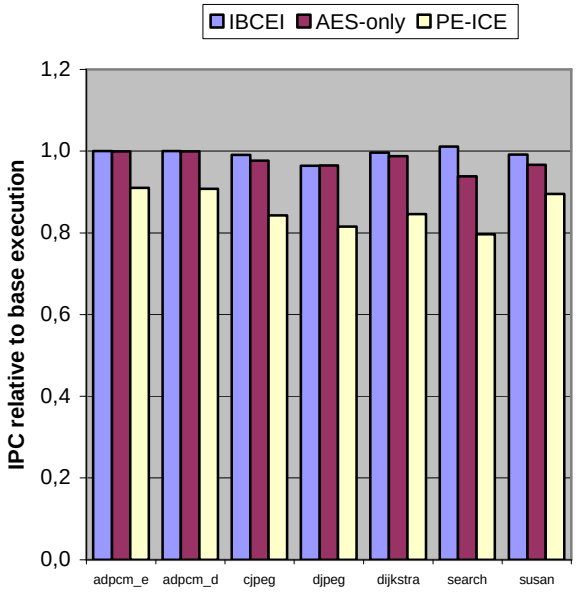

Fig. 9. Performance for the 8k i-cache

Figure 8 and Figure 9 show the Instruction per Cycle obtained for each benchmark, normalized to the base execution. Note that in all cases the IBC-EI outperforms the AES-only approach, while providing integrity checking in addition to encryption. Moreover, the IBC-EI also outperforms the original PE-ICE implementation.

Note that, as the cache size increases, the performance tends to be the same as the original. This fact is due to the nature of cache misses that tends to be Compulsory in majority, especially for the smaller benchmarks like ADPCM. Once in the cache, and with a low miss ratio, the Decompressor engine is not more used, and the only AES usage is for fulfilling the cache.

On average, IBC-EI performance penalty is negligible (less than 1%) when compared to a base execution.

5 Improvement Discussion

The blobk-level AREA concept and the related engine for SoC, PE-ICE, optimize run-time performance and hardware resources when compared to conventional method [3, 4, 11] providing data confidentiality and integrity [5, 6]. However, PE-ICE as well as those conventional methods still generates a non-negligible off-chip memory overhead to store off-chip the tag required for memory integrity checking. We show that our proposed engine IBC-EI, which adds a layer of code compression to PE-ICE, allows for the cancellation of this off-chip memory overhead.

Moreover, in term of run-time performance, IBC-EI outperforms the first by about 15% in average in terms of IPC, for the same platform and benchmarks. In general the IBC-EI provides the means to use more effectively the expensive AES block, providing not only one cache line per chunk. As a result, IBC-EI allows for providing code integrity in addition to code confidentiality at almost no cost when compared to an AES encryption scheme. On the oher hand, the internal memory necessary to implement it will be greater than the one for PE-ICE, due to the dictionaries storage.

In terms of security, our solution can be seen as an extension to the PE-ICE solution which relies on the diffusion feature of the encryption algorithm. Thus corrupting one bit of the ciphertext will affect various bits in the plaintext after decryption. In other words, changing one bit in the memory will affect the tag area with a strong probability in the decrypted chunk and invalidate the block after the tag matching process. This probability depends on the length of the tag as shown in [6]. When the 32-bit address is used as tag, spoofing attack has $1/2^{32}$ likelihood to succeed [6]. Concerning splicing attacks, the tag used – the chunk address – is a nonce, thus a different tag is used for every chunk stored off-chip. It results that a spliced block will always be detected upon the corresponding tag matching process. On the other hand, AES encryption is implemented in IBC-EI, preventing passive attacks carried out on the processor-memory bus to succeed. Moreover, compression will enhance the entropy before the encryption, which in turn, can provide better statistical spread of information in the ciphertext.

6 Conclusion

In this paper we presented IBC-EI, a code compression method tailored for integrity checking and confidentiality. We showed that the code

compression layer of IBC-EI added to PE-ICE allows for the reduction of its off-chip memory overhead. Moreover, IBC-EI incurred a negligible run-time performance hit. It results that IBC-EI ensures the confidentiality and integrity security services to code at almost no cost.

Future work involves the relocation of cache lines to better explore the chunk space and/or to better compress the critical execution paths of the code. The energy measurements are also necessary to provide a complete picture of the solution. Moreover, we currently adapt the concept behind IBC-EI to read write data.

7 References

[1] M. G. Kuhn, "Cipher Instruction Search Attack on the Bus-Encryption Security Microcontroller DS5002FP", IEEE Trans. Comput., vol. 47, pp. 1153–1157, October. 1998.

[2] A. Huang. "Keeping secrets in hardware the microsoft xbox case study". MIT AI Memo, 2002.

[3] Gookwon Edward Suh, "AEGIS: A Single-Chip Secure Processor", PhD thesis, Massachusetts Institute of Technology, September 2005.

[4] G. Suh, D. Clarke, B. Gassend, M. van Dijk and S. Devadas, "Efficient Memory Integrity Verification and Encryption for Secure Processors" In *Proceedings of the 36th Int'l Symposium on Microarchitecture (MICRO-36)*. pp. 339-350, (Dec. 2003).

[5] 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 43rd Design Automation Conference (DAC-43)*. pp. 506-509, (July 2006).

[6] R. Elbaz, "Hardware Mechanisms for Secured Processor-Memory Tansactions in Embedded Systems" *PhD Thesis, LIRMM laboratory Montpellier University*, 2006.

[7] A. Wolfe and A. Chanin. "Executing compressed programs on an embedded RISC architecture". In *Proceedings of the Int'l Symp. on Microarchitecture* pp. 81-91 (Dec. 1992).

[8] Gaisler, G. Leon, 2003. *Available at: http://www.gaisler.com accessed June/2006*

[9] Guthaus, M., Ringenberg, M., Ernst, D., Austin, T., Mudge, T. and Brown, R. MiBench: a free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization* pp. 3-14, (Dec. 2001).

[10] Lee, C., Potkonjak, M. and Mangione-Smith, W. MediaBench: a tool for evaluating and synthesizing multimedia communication system. In *Proceedings of the Int'l Symp. on Microarchitecture*, pp.330-337, (Dec. 1997).

[11] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell and M. Horowitz. "Architectural Support for Copy and Tamper Resistant Software". In *Proceedings of the 9th Int'l Conference on Architectural Support for Progamming Languages and Operating Systems (ASPLOS-IX)* pp. 168-177, (Nov. 2000).

[12] M. Bellare and C. Namprempre, "Authenticated Encryption: Relations among Notions and Analysis of the Generic Con-struction Paradigm", In *T. Okamoto, editor, Asiacrypt 2000, volume 1976 of LNCS, p. 531-545. Springer-Verlag*, Berlin Germany, December 2000.

[13] C. Fruhwirth, "New Methods in Hard Disk Encryption", Institute for Computer Languages, Theory and Logic Group, Vienna University of Technology, 2005. *Available at http://clemens.endorphin.org/cryptography. Accessed May/2007*

[14] National Institute of Science and Technology. FIPS PUB 197: Adavnced Encryption Standard (AES), Nov. 2001

[15] R. Azevedo, "An architecture for Executing Compressed Code in Embedded Systems" *PhD Thesis, Institue of Computing, UNICAMP*. 2002.

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