19th International Symposium on Computer Architecture and High Performance Computing

A Code Compression Method to Cope with Security Hardware Overheads

Eduardo Wanderley
CEFET-RN, Brazil
wanderley@cefetrn.br

Romain Vaslin, Guy Gogniat, Jean-Philippe Diguet
LESTER, UBS, France
[vaslin,gogniat,diguet]@univ-ubs.fr

Abstract

Code Compression has been used to alleviate the memory requirements as well as to improve performance and/or minimize energy consumption. On the other hand, implementing security primitives on Embedded Systems is always costly in terms of area and performance. In this paper we present 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 transactions. The principle is to keep the code compressed and ciphered in the memory, thus reducing the memory footprint and providing more information per memory access. For the Leon processor and a set of benchmarks from the Mediabench and MiBench suites the habitual overheads due to security trend to zero in comparison to a system without security neither compression.

1. Introduction

The demand for embedded system has been growing more and more and the mobile terminal is one of the faces of this astonishing increase. Ubiquitous bank transactions, for example, are now a real requirement for this type of embedded system. Despite of the energy consumption, performance and miniaturization axis to drive the designs, the security requirements play nowadays an important role in the system conception.

Threats come from everywhere and one possibility is probing the processor-memory traffic in order to interfere in the execution flow and/or obtaining secret information, like passwords or confidential data [1,2]. These board-level attacks challenge not only data confidentiality but also data integrity.

While memory encryption can be used to ensure confidentiality of processor-memory transactions, data integrity requires additional information to be added to the original data in order to verify its integrity. These meta-data called TAGs can be computed with MAC – Message Authentication Code – functions [3], or are nonce (redundancy) added to plaintext block before block encryption [4]. The main drawback of these approaches is the off-chip memory area overhead, due to the TAG storage [4, 5, 6, 7].

Code Compression was first idealized to respond to the memory constraints of embedded systems. When introduced [8], a decompressor hardware engine was used between the cache and the main memory of a system, therefore the decompression overhead could be hidden by the cache (a block decompression occurs only on a cache miss). This scheme imposes a penalty in execution time due to decompression. On the other hand, less access to the main memory is expected, which in turn, alleviate the performance problem.

In this paper we devise the Instruction Based Compression with Encryption and Integrity checking, IBC-EI, method. IBC-EI uses the concept of block-level Added Redundancy Explicit Authentication – AREA [4, 7] – to pursue integrity and confidentiality at the same time, while has its bases founded on the Instruction Based Compression, IBC, method [9]. The objective is to keep the code compressed and ciphered in the main memory. Then, during execution, the block is deciphered, exposing a TAG and the compressed information. While the decompressor retrieves the original instructions from a dictionary, the TAG is checked to guarantee integrity.

This work focuses on the protection of the code (i.e. Read Only information), thus it is possible to compress and cipher off-line. On the other hand, decompression and deciphering are done on-the-fly. For performance reasons, the deciphering and the decompression are performed in hardware.

This paper describes the compression method. It is organized as follows: in Section 2 the threat model is described. Section 3 presents the proposed code compression method. Section 4 shows the results for the Leon Processor [10] and a set of benchmarks from MiBench [11] and Mediabench [12] suites; in the Section 5 a discussion is presented. Finally, we present our conclusions and future work in Section 6.

1550-6533/07 $25.00 © 2007 IEEE
DOI 10.1109/SBAC-PAD.2007.40

IEEE computer society

Authorized licensed use limited to: UNIVERSITE DE BRETAGNE SUD. Downloaded on July 28, 2009 at 08:35 from IEEE Xplore. Restrictions apply.

2. Threat Model and Existing Countermeasures

2.1. Threat Model

In this work we assume some constraints in the threat model. The most important is to consider the SoC as secure, including the processor, the caches and the hardware modules used in our solution. The attacks to refrain from are those threatening the off-chip memory and the bus transfers. Both confidentiality and integrity are challenged. An attacker may want to retrieve sensitive data (confidentiality menace) or disturb the program execution (integrity menace).

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. We do not face replay attacks (temporal permutation of memory block) because we deal only with Read Only information (the code).

2.2. Existing Techniques for Encryption and Integrity Checking

In order to prevent the attacks before mentioned we must provide data integrity and confidentiality.

Data integrity can be reach by using a MAC that generates a compact representative image of the information. This image is called TAG and is supposed to be kept along with the information, as a signature. 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.

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. Encrypt-then-MAC is the most implemented scheme since proved secure [13]. However, in all cases during the ciphering and/or during the deciphering, the MAC and Encryption or Decryption units are not fully parallelizable, producing an impact on performance and/or on silicon area in the implementation.

Thus, in this paper, we will use the concept of block-level AREA introduced in [4, 7]. 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 [14]: 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.

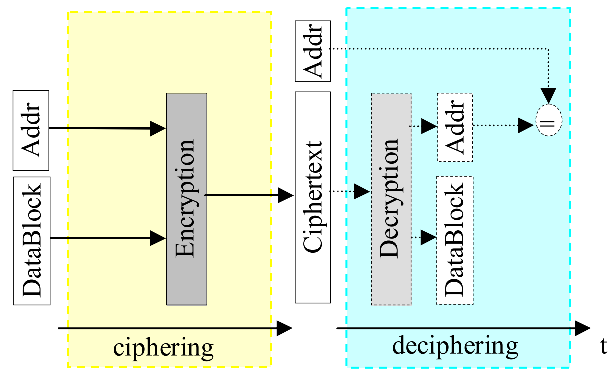


Fig. 1. Parallelized confidentiality and integrity checking scheme.

The decryption is performed using a key securely stored on the SoC. The SoC decrypts the block it fetches from memory and verifies that the first 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.

The first implementation of the block-level AREA concept, called PE-ICE was proposed in [7] and is shown in Figure 1. We use the address of each plaintext block as 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.

Nevertheless, this approach will still incur performance penalties and, more inconvenient, a considerable off-chip memory overhead. 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.

Authorized licensed use limited to: UNIVERSITE DE BRETAGNE SUD. Downloaded on July 28, 2009 at 08:35 from IEEE Xplore. Restrictions apply.

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

Code compression is first 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. Unfortunately, the decompressor used to restore the original information will introduce some penalty in performance. However, this penalty can be hidden by the expensive memory operations reduction.

The idea to derive a code compression method integrating encryption and integrity checking 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 2. First the original code is compressed and then encrypted. 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 Processor.

The Decryptor and the Decompressor are allocated into the trusted zone and are used only upon a cache miss. 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 [9]. The index size, IDXsize, is a function of the number of dictionary entries, DICTentries, as show in Equation1

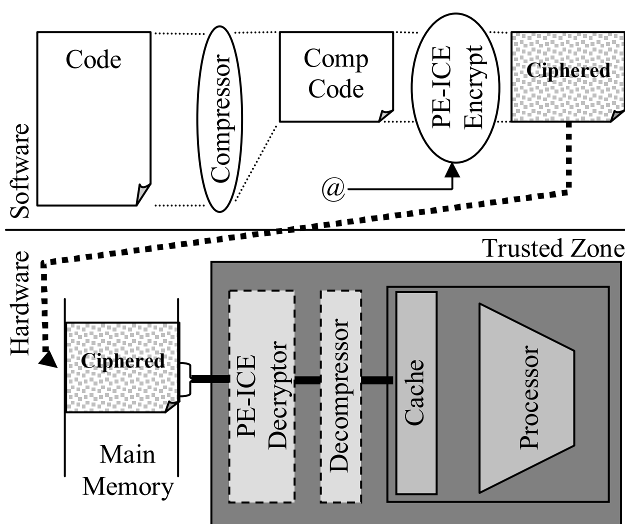


Fig. 2. Code Compression and security

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

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 is sometimes presented in percentage.

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

This simple approach is still insufficient because the code execution 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 blocks 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 [9]. 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.

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 3. Usually, 4 dictionaries with 2 bits prefixes will lead to better compression of the code. Then, we will use a set of four dictionaries for our experiments.

Authorized licensed use limited to: UNIVERSITE DE BRETAGNE SUD. Downloaded on July 28, 2009 at 08:35 from IEEE Xplore. Restrictions apply.

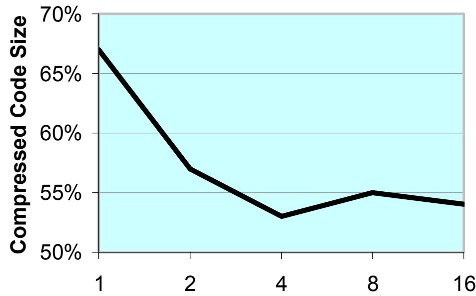


Fig. 3. Compressed Code size for multiples dictionaries usage relative to the original

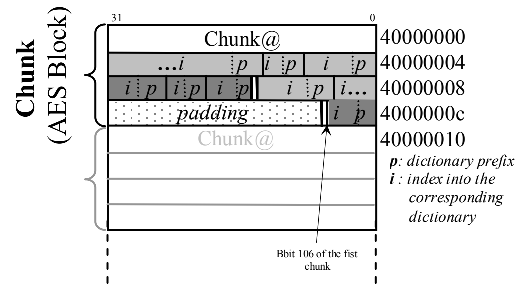
3.1. Integrating Confidentiality and Integrity

In our description we use the term chunk to indicate the atomic IBC-EI block loaded from memory for decryption and integrity checking. A chunk, shown in Figure 4, 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 a block cipher, in the case the Advanced Encryption Standard [15] – AES, and stored in the main memory. 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 consecutive time-expensive AES decryptions to fulfill a simple 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 corresponding compressed line is present. Moreover, as more than one cache line can be accommodated into the payload, every cache line needs an offset identifier, which represents the starting bit of compressed cache line, relative to the IBC-EI Block start

Figure 4 shows a conceptual view of a chunk in the memory (before encryption). Such a chunk contains two compressed cache lines. In this case, the original cache line size is 16 bytes long. 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.

It is also possible to observe the ATT architecture. Note that every original cache line address can be identified by only a subset of the address bits. The number of bits in this subset is program size dependent. For example, a 1k bytes code will have 64 16-byte cache lines associated. Thus observing 6 bits in the address it is possible to identify every cache line. We use these bits as an index into the ATT.



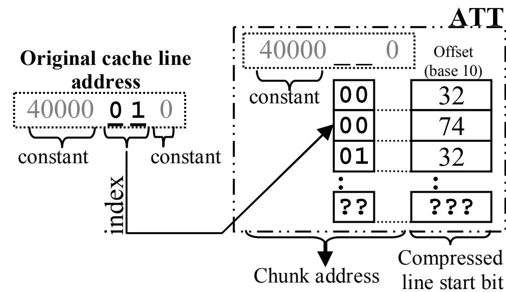


Fig. 4. IBC-EI Block and ATT architectures

The same principle is used to keep the corresponding chunk addresses in the ATT. Nevertheless, two original cache lines that have their corresponding compressed cache line in the same chunk will have two entries in the ATT with the same IBC-EI block address, but different offsets.

The offsets have fixed sizes as they represent the starting bit of a compressed cache line relative to the bit 0 of the chunk. The offset size depends only on the chunk size. On the other hand, the ATT size depends also on the original cache line size.

In Figure 4 we observe two original 16 bytes cache lines compressed in the same chunk. The first one occupies bits 32 to 73 of the chunk. The second one occupies bits 74 to 106. Note that the ATT holds one entry for every original cache line with the same chunk address, but with two different offsets (32 and 74). The compression algorithm is responsible for creating the adequate ATT for a given program.

3.2. The Compression Algorithm

The compression algorithm used to generate the compressed code, the ATT and the Dictionaries is composed by two main routines: `CreateDictionaries` and `IBC-EICompress`. `CreateDictionaries` receives the Original Code and produces the 4 dictionaries to be used by `IBC-EICompress`. To do this, we rank the instructions by usage and divide in four groups with the same contribution. When this division is not exactly we use a branch-and-bound approach in the neighborhood of the division thresholds to find the best partition, the one that yields the best compression.

Authorized licensed use limited to: UNIVERSITE DE BRETAGNE SUD. Downloaded on July 28, 2009 at 08:35 from IEEE Xplore. Restrictions apply.

```
01 IBC-EICompress(c, // Original Code
02                b, // IBC-EI BlockSize
03                l) // Original Cache Line Size
04 {
05   d  ← CreateDictionaries(c);
06   cl ← NULL; //compressed cache line
07   bb ← TAG_Size; // current block bit (used-1)
08   cb ← CreateNewBlock();//current block
09   for (ol: each original cache line in c) do {
10     cl ← CompressCacheLine(ol, d);
11     if (bb + size(cl) > b){ //comp cache line
                              does not fit current block
12       FulfillPadding(cb, bb, b-1);
13       cb ← CreateNewBlock();
14       bb ← TAG_Size;
15     }
16     AddCompLine(cb, cl); //cl is assembled in cb
17     ATT ← CreateNewEntry(cb, bb);//bb=offset
18     bb ← bb + size(cl);
19   }
20 }
```

Fig. 5. IBC-EI Compression Pseudo-Code

IBC-EICompress receives the original code, the chunk size and the cache line size as input parameters, as shown in the Figure 5. The first step in the compression algorithm is exactly calling the dictionary routine. Then, the first IBC-EI block is created. Now, for every cache line of the original code we try to insert the corresponding compressed line into the current block. The CompressCacheLine routine is responsible for transforming, using the dictionaries information, an original cache line into a sequence of prefixes and indexes. If the recent assembled compressed cache line does not fit the block in use, this latter will be padded, and a brand new block is created. Then the compressed line will be added.

At this point, it is possible to identify which block holds the current compressed cache line, and exactly where it starts (*bb* variable in the pseudo-code). These are the information we need for a new ATT entry.

There is just one usage detail of the present algorithm: if one compressed cache line does not fit an empty payload part of a block. For this situation, omitted for the sake of simplicity in the pseudo-code, the compressed cache line is necessarily split into two consecutive blocks.

3.3. The Decompression Hardware

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^{-1} deciphering unit 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

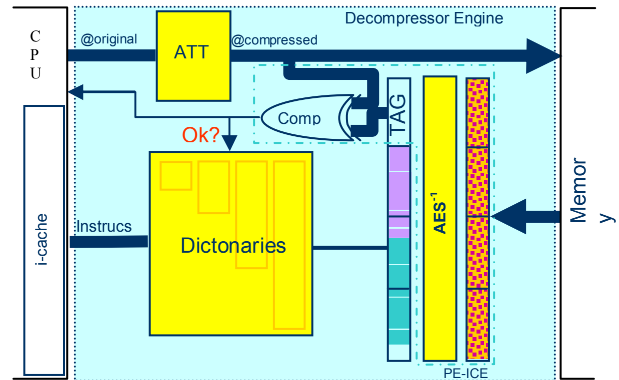


Fig. 6. Decompressor Engine

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 cache/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, thus, no memory access is required, neither the AES^{-1}.

Note that the expensive deciphering AES^{-1} operation 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.

4. Results

We have run extensive simulations in order to evaluate our proposal. The base setup is shown in table 1. Then we varied the I-cache size and line size. We observed the memory latency impact on our design. We also increased the IBC-EI block to check the effects. Finally we noticed the impact of choosing different dictionaries based on the execution frequency of the each instruction instead of on the static count. The following results are presented relative to an implementation without security primitives neither compression.

Table1: Simulator Architectural Parameters

Parameter	Specification
I-Cache	1k / 16B lines, Direct Mapped
D-Cache	8k / 16B lines, Direct Mapped
Memory latency	75 cycle per cache line fetch
Memory bus frequency	133MHz
Processor frequency	400MHz
IBC-EI Block	128 bits
AES latency	17 cycles
Decompression latency	4 cycles

Authorized licensed use limited to: UNIVERSITE DE BRETAGNE SUD. Downloaded on July 28, 2009 at 08:35 from IEEE Xplore. Restrictions apply.

4.1. Compression Ratio Analyses

Initially we observed the compressed code formation for the basic setup. Figure 7 shows the results. The first remark in this figure is that the IBC-EI blocks that will be stored in the off-chip memory represent on average 53% of the original code size. On the other hand, it is also necessary to keep the ATT and Dictionaries contents. They are stored in the decompressor engine. If we consider the area overhead expressed as the compression ratio, comprising also the ATT and the Dictionaries, the results point to 101% on average relative to the original. The djpeg has the poorest compression ratio, but the overhead is still below 20%. If we consider that for every cache line we added a 32-bit TAG (25% of a cache line size), in all the cases the compression solved the overhead problem due to the security implementation. Many parameters influence the compression ratio of IBC-EI. The cache line size, the IBC-EI block size and the dictionary construction will change the results.

Varying the cache line size is important to present the results for several available configurations of real SoC implementations. From Table 2 we see that increasing the cache line will always produce better compression. In fact, increasing the cache line size will produce smaller ATTs because the number of cache line in the original code (i.e. the number of ATT entries) will decrease. If we double the cache line size we will half-size the ATT. The number of padding bits will also decrease, because the granularity of the fragmentation will increase.

The size of the IBC-EI block is also important. One of the code compression benefits is that we can minimize the use of the AES^{-1} hardware unit. Although the AES works with a fixed size data block (128 bits), the Standard was defined after a contest in which the Rijndael proposal was chosen to be the standard. This proposal is not limited to 128 bits, thus 192 and 256 bits are also possible. As the IBC-EI uses the AES to decipher its blocks, we also investigated the possible 256 bits block size originally proposed.

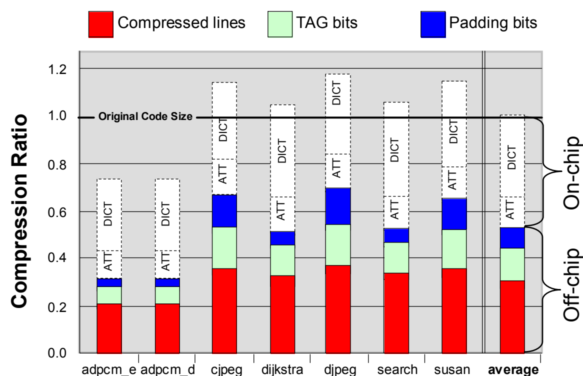


Fig. 7. Compressed code components

Table 2: Compression Ratios (%)

Dictionary	Static				Dynamic			
Block Size (bits)	128		256		128		256	
Cache Line Size (bytes)	16	32	16	32	16	32	16	32
adpcm_e	73	69	66	64	78	75	71	69
adpcm_d	72	68	66	64	77	74	70	67
cjpeg	115	96	94	90	128	104	99	98
dijkstra	105	93	95	92	119	102	99	94
djpeg	118	99	96	92	135	108	101	99
search	106	94	97	94	119	103	99	95
susan	115	98	96	92	132	107	98	92
Average	**101**	**88**	**87**	**84**	**113**	**96**	**91**	**88**

When the block size increases, more cache line can be accommodated in the payload thus reducing the padding bits used. That is why we observe that increasing the block size we produce better compression ratios. Using a 256-bit block and 32 bytes cache lines will produce the best results for compression for the static dictionary.

The static dictionary is the one that we build based on the static profiling of the code. The motivation to build dynamic dictionaries, in a sense that these dictionaries are built based on the execution profiling of the code, is that the code execution hot spots will have the smaller compressed instructions (the pairs (prefix, index)). In this case, the blocks that contain the most frequently executed instructions will accommodate more compressed cache lines. During execution these blocks will be more frequently invoked by the processor, increasing also the probability of finding the compressed line already deciphered and ready to be decompressed inside the IBC-EI engine.

Unfortunately, using dynamic dictionaries will not maximize the code compression because the instructions that are executed the most are not the same instructions we find more frequently in the code. That is the phenomena we observe in our table 2. For the same block and cache line parameters, using dynamic dictionaries will always produce worst compression ratios. On the other hand, using dynamic dictionaries is supposed to provide better performance results.

4.2. Performance Analyses

The goal of the following experiments is to show the parameters that affect the performance of a system based in our proposal. All the performance results are presented, in terms of IPC, relative to the execution of the program without security neither compression (base execution). We have evaluated IBC-EI varying the memory performance; the cache setup, including size and line size; and the dictionary construction aspects. While varying one of the parameter, the others remain as stated in Table 1.

In the first observation we proposed some memory setups. As we considered the AMBA bus running at 133MHz and the Processor running at 400MHz every bus clock takes 3 processor clock cycles. We evaluated this question by using 6, 15 and 25 bus cycles to

Authorized licensed use limited to: UNIVERSITE DE BRETAGNE SUD. Downloaded on July 28, 2009 at 08:35 from IEEE Xplore. Restrictions apply.

retrieve a block from the main memory. The reasoning is that a 6 cycle memory is very fast and a 25 cycle (75 processor clock cycles) is too slow.

In the Figure 8 we present our results. Observe that for fast memories, the impact on performance due to the IBC-EI engine reaches about 15% on average, which means that the price for security will be the 15% performance hit. On the other hand, as the memory becomes slow, the time to bring a block to the IBC-EI engine becomes more and more dominant. For a slow memory, the performance hit is only 1% on average and in some cases is really possible to execute faster then the base execution. For the intermediate speed memory the performance hit is only 5%, which means that if we do not use a very fast memory the performance lost is fast mitigated by our proposal.

The ADPCM benchmarks are not very influenced by the memory performance. As they are tiny codes, the cache holds all the instructions for the execution, so no memory access is used (except the ones generated by the compulsory cache misses).

The memory speed is not relevant to the compressibility of the code. Neither is the cache size. Nevertheless, the cache line size interferes on the compression ratio, as we have already observed. The question is: would the cache line size interfere also in performance? To answer this question we varied this parameter from 16 to 32 bytes.

The expected performance speedup due to the cache size increase can be observed in Figure 9. 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.

The analysis for the cache line size is a little bit different. In fact, augmenting the cache lines size will allow us to explore more spatial locality. This is very code dependent. But another fact is also important: we have already seen that increasing the cache line will produce better compression ratios. In this case, the compression of the code is also responsible to the performance improvement.

This opens a new question: will the compression play an even more significant role in performance? To investigate this possibility we tried the following approach: if the trace that is executed the most is also the smaller, more instructions will be fetched into the processor per memory access. On the other hand we can not guarantee that the compressibility will be the same as before. Then, we constructed the dictionaries dynamic profiling of the code execution. Figure 10 shows the positive impact of this approach. On average a 10% performance benefit can be observed. This represents a trade-off as compression will be reduced by using dynamic dictionaries. Nevertheless, from the execution point of view, the code will appear to be smaller. Thus, the system constraints will determine the balance between performance and area, which will drive the choices for the systems parameters.

Finally, to provide some information about the quality of our results in comparison to other approaches we have implemented the PE-ICE solution and the AES-only solution. The first will provide integrity and confidentiality but will produce the

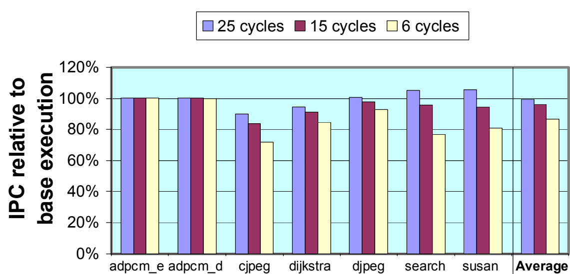


Fig. 8. Memory speed dependent performance

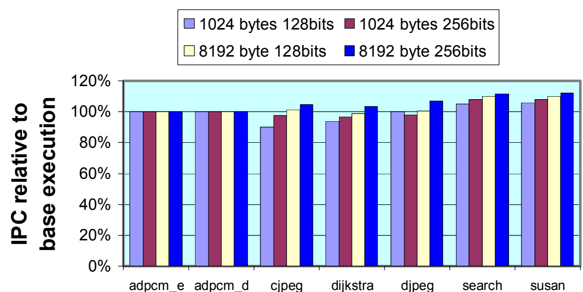


Fig. 9. Cache Setup dependent performance

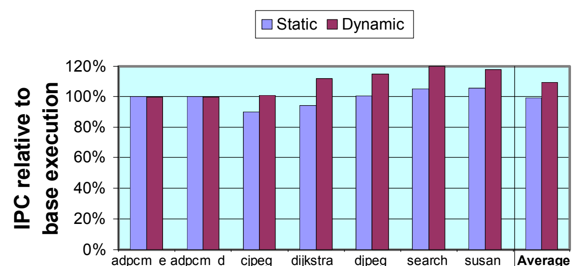


Fig. 10. Dictionaries dependent performance

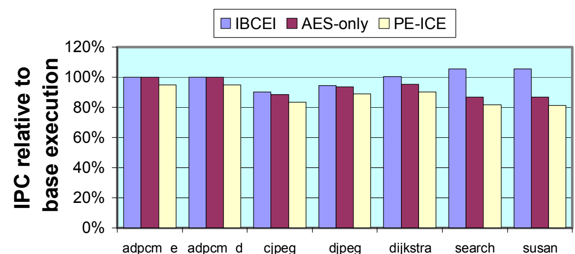


Fig. 11. Results comparison

Authorized licensed use limited to: UNIVERSITE DE BRETAGNE SUD. Downloaded on July 28, 2009 at 08:35 from IEEE Xplore. Restrictions apply.

impact of 25% in code area and some extra cycles to retrieve the TAG. The AES-only has not code area overhead, but it will provide only confidentiality. The performance for these solutions can be seen in Figure 11. Note that in every case the IBC-EI outperforms both AES and PE-ICE.

5. Improvement Discussion

The block-level AREA concept and the related engine for SoC, PE-ICE provides data confidentiality and integrity [4, 7]. However, PE-ICE, as well as those conventional methods, still generates a non-negligible off-chip memory overhead to store the TAG required for memory integrity checking. We show that our proposed engine IBC-EI allows for the cancellation of this off-chip memory overhead.

In general the IBC-EI provides means to use more effectively the expensive AES block, by supporting 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 other hand, the internal memory necessary to implement it will be greater than the one for PE-ICE, due to the dictionaries

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 mostly 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 [4]. When the 32-bit address is used as TAG, spoofing attack has $1/2^{32}$ likelihood to succeed

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. Moreover, AES encryption is implemented in IBC-EI, preventing passive attacks carried out on the processor-memory bus to succeed.

Finally, 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.

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

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

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

[6] 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).

[7] 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).

[8] 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).

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

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

[11] 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).

[12] 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).

[13] 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.

[14] C. Fruhwirth, “New Methods in Hard Disk Encryption”, Institute for Computer Languages, Vienna University of Technology, 2005. May/2007. Available at http://clemens.endorphin.org/cryptography.

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

Authorized licensed use limited to: UNIVERSITE DE BRETAGNE SUD. Downloaded on July 28, 2009 at 08:35 from IEEE Xplore. Restrictions apply.