BUS DECRYPTION OVERHEAD MINIMIZATION WITH CODE COMPRESSION

Eduardo Wanderley [*]

Dept. of Informatics
CEFET-RN
1559 Salgado Filho, Natal 59015 RN Brazil
email: wanderley@cefetrn.br

Guy Gogniat, Jean-Philippe Diguet

LESTER Laboratory
University of South Britany
rue Saint-Maude, 56100, Lorient, France
email: [gogniat, diguet]@univ-ubs.fr

ABSTRACT

Code Compression has been shown to be efficient in minimizing memory sizes for embedded systems as well as in power consumption reduction and performance improvement. In this paper we claim that code compression is also able to minimize the performance penalty of bus encryption schemes, where a ciphered program flows through a deciphering unit and reaches the processing unit. The result on performance for the Leon Processor, using a set of benchmarks from MediaBench and MiBench suites reveals that the AES deciphering unit can be used more rarely (50% less) and performance is never degraded comparing to the original secure system.

1. INTRODUCTION

Ubiquitous computing is coming into a reality thus making security a serious issue for mobile information transactions. Embedded systems have been used in a large myriad of applications including security systems.

In this scenario, we found the processor-memory information (data+program) traffic as a weakness in security. Observing memory contents flows, to and from the processor, reversing engineering of the software and/or access to private or sensitive data can be accomplished.

The choice for guarantying secure bus transfers is using a ciphering algorithm to encode the memory footprint of the embedded software. Then, a deciphering scheme must take place to permit the software usage. This model does not come true without a cost, especially in execution time. The penalty of the deciphering unit, in hardware or in software, imposes a performance loss often non-negligible.

Code compression (CC) was first idealized to respond to the memory constraints of embedded systems. When first introduced [1], a decompressor engine was located between the cache and the main memory of a system (CDM, Cache-Decompression-Memory architecture), 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, alleviates the performance problem.

[*]Post-doc at LESTER lab, University of South Britany

Recent trends in code compression [2, 3, 4] points to placing the decompression engine between the cache and the processor (PDC, Processor-Decompression-Cache architecture). In this scheme, the cache holds compressed code and less accesses to the main memory is required. Nevertheless, the decompressor is placed in the processor critical path. This imposes the decompressor to be very fast, and some of the outstanding compression algorithms (normally with slow decompression associated) are discarded. The benefits of better cache hit ratio use to outperform the decompression overhead and, in some occasions, the decompressor can be attached to the original processor pipeline thus providing no decompression penalty [5]. This scheme leads to performances improvement as well as power consumption reduction.

In this paper we present a scheme where the embedded software is compressed and then encrypted. During run-time a decipher hardware unit retrieves the compressed code that will be stored in the cache. Then, the decompression engine delivers to the processor the required instruction.

The assumption is that the global decipher overhead will be overcome by the higher cache hit ratio (less accesses to the main memory and thus to the deciphering unit).

This paper is organized as follows: Section 2 addresses the compression and ciphering schemes; Section 3 outlines the system overview; in section 4 we provide some results; following, in Section 5, we address some comments and discussions. We finalize with our conclusions in Section 6.

2. COMPRESSION AND ENCRYPTION MODELS

The ComPacket is a code compression method that fits in a class of methods referred as dictionary-based, where compression is achieved by replacing instructions in the code for indexes into a dictionary. The compressed unit is called *ComPacket* (*Compressed Packet*), a 32-bit word with a set of indexes to a 256 entries incomplete dictionary. The method

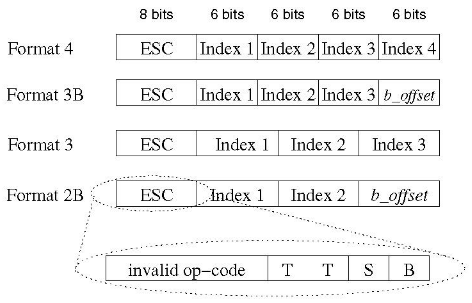

Fig. 1. ComPacket Formats

supports four ComPacket formats, as delineated in Figure 1. Format 4 has 4 indexes of 6 bits such that they can only access the first 64 instructions of a dictionary. Format 3 has 3 indexes of 8 bits, allowing full access to the dictionary. Format 3B has 3 indexes of 6 bits and a branch slot of 6 bits such that the ComPacket admits one branch instruction (only branches with offsets that can be represented with 6 bits are eligible). The Format 2B has 2 indexes of 8 bits and a branch slot of 8 bits, also allowing one branch instruction (in this case, branch offsets that can be represented with 8 bits are eligible).

An invalid opcode is used to identify a ComPacket (in the SPARC ISA, op=00_2 and op2=$00X_2$). *S*=0 denotes a ComPacket with 8-bit indexes, while *SB* and *TT* are used to identify the type and contents of each compressed packet.

After building the dictionary, the next step is to compress the code by assigning the ComPackets, according to the explained encoding and looking for the best compression as possible for the method. A greed algorithm is used to assign the ComPackets and rebuild the code. A complete reference of the ComPacket can be found in [4];

For the ciphering, we used the Advanced Encryption Standard [6], which is a symmetric-key encryption algorithm approved by the National Institute of Standards and Technology (NIST). The AES uses blocks of 128bits and keys of 128bits, 192bits or 256bits. A set of transformations, necessary to a round of the algorithm, can be efficiently implemented with lookup tables. 10 to 16 rounds are necessary to encipher a block. A parallel key expansion algorithm is used to produce an arborescence of the key which is used in chucks of 128bits in each round. Deciphering is basically using the inverse transformations and can be also implemented using lookup tables.

The choice for these two methods is based on the availability of the compression method for the platform, and the wide use of the ciphering algorithm, although the general idea can be used and proved with other combinations of methods.

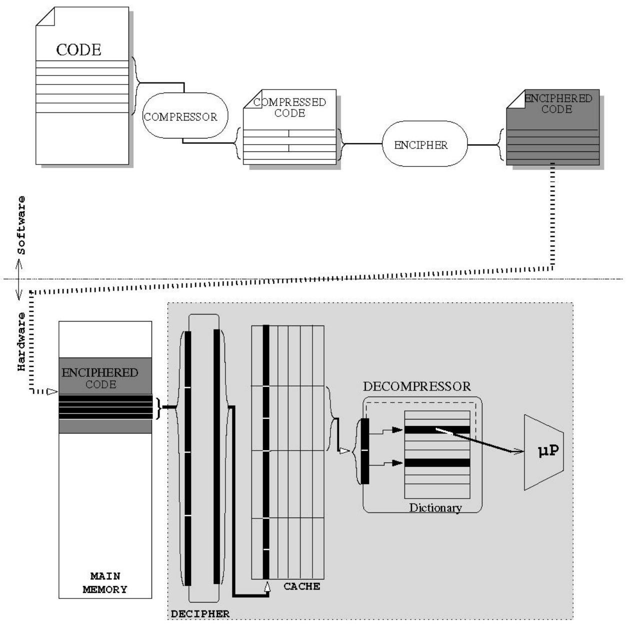

Fig. 2. Proposed Architecture

3. SYSTEM OVERVIEW

The proposed implementation will deal with problem of memory encryption, in which a decryption unit is positioned in a secure area of the processor, between the cache and the main memory, so that, by observing the bus activity an attacker will find encrypted information. This approach will impose an overhead in the processing time due to the deciphering unit.

Using code compression the number of transfers between the cache and the main memory will decrease, which means that the global decryption overhead will be minimized, and if using a scheme that naturally improves performance the overhead can be completely avoided.

The general idea is presented in Figure 2. After the compression phase, the AES algorithm is used to Encrypt the blocks of 4×32bits (the same size of a cache line). The compressed and encrypted code is then loaded into the main memory. When the processor asks for an instruction, the decompressor asks the cache if it is available. In such case, no deciphering is necessary. If the instruction in the cache is a codeword the decompressor acts delivering the corresponding instructions. If the instruction is not present in the cache, the deciphering unit is used, retrieving the block from the main memory and converting it into a compressed code that is delivered to the cache. This method relies on the efficacy of the compression to avoid some cache misses and, consequently, accesses to the main memory. Moreover, the density of the code that is transferred through the bus is higher, so that, less deciphering is used for the whole execution of the code.

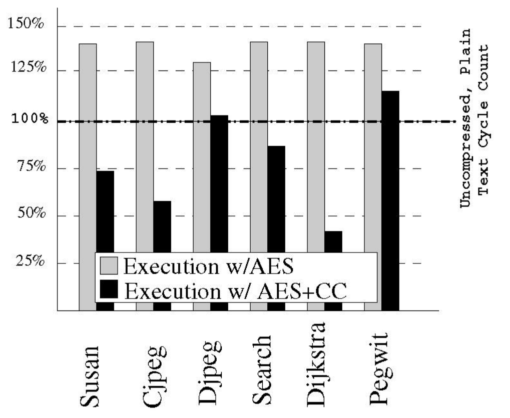

Fig. 3. Cycle Count Relative to Uncompressed, Plain Text Code Execution

4. RESULT

In order to validate the proposed scheme we simulated a set of benchmarks from Mediabench [7] and MiBench [8] suites comprising *Search*, a string search algorithm; *Susan*, used to smooth image corners; *Dijkstra*, an algorithm used to find paths in a graph; *cjpeg* and *djpeg*, image conversion algorithms; and *Pegwit*, an encryption program;

All the codes were compiled with a GCC cross compiler for the Leon (Sparc V8) processor [9]. The main memory was chosen to be unlimited in terms of size, and the access time will cause the processor to wait 80 clock cycles to receive the corresponding required cache block (miss penalty), like in [10]. The cache sizes were chosen by simulating the original application and finding the sizes in which a double increase in size will not produce a significant hit ratio improvement ($\leq 5\%$). The decompressor unit was designed so as to be coupled with the cache controller and does not implies any overhead to decompress an instruction [5].The AES deciphering unit was estimated like in [10] in which a block requires 40 clock cycles to be deciphered.

Figure 3 shows the cycle count relative to the original execution of the code with the same simulation framework. For the execution with a simple AES encryption, considering the deciphering unit overhead, on average we obtain 39% of performance decrease, while using code compression and AES together the performance will increase in 20%. In all cases the AES+CC outperforms AES.

The compression ratios of the used applications are shown in Table 1. The compression ratio is defined as the memory footprint size (this includes the dictionary size) relative to the original uncompressed code. The clock cycles in the AES deciphering computation is also depicted, relative to the original number, which means that Dijkstra, for example, will need only 20.55% of the cycles used originally in the deciphering unit.

Application	Compression Ratio	Clock Cycles in AES
Cjpeg	92%	32.80%
Djpeg	98%	70.30%
Dijkstra	96%	20.55%
Pegwit	86%	80.20%
Search	88%	56.42%
Susan	86%	45.25%

Table 1. Compression Ratio and AES Clock Cycles amount due to Compression

Normally, the best is the compression, the best is the performance result, but in some cases this does not happen. Dijkstra, for example, have a poor compression ratio, but a very effective performance using compression. This comes from the fact that most of the ComPackets formed in the code belongs to the main execution code traces, and as a famous rule of the thumb establishes, 90% of the execution time is located in 10% of the code. On the other hand, Pegwit is clearly the contrary, which means that the majority of the ComPackets are outside the critical execution path.

5. DISCUSSION

The proposed model brings to the problem of the overhead caused by a deciphering unit, a solution that benefits not only the performance but also permits to save memory area in all cases and even to get some performance improvement. In fact, when using a secured code, some performance penalty is expected, but the code compression copes with the problem and produces a speedup of 1.2 on average.

This is a desired triad in the embedded system field. In addition, the four axes, energy aspects, can be also foreseen as positive, as less external memory accesses are required and thus less AES computation.

The decompressor area is absolutely small. It was implemented in a FPGA and added only 6,473 equivalent gates to the original 1,923,763 Leon processor implementation. The AES deciphering unit implementation will remain the same independent of the code compression usage, so that the area overhead depends on the implementation design.

In the security point of view, compression imposes a noise in the expected results from an attacker. Execution time as well as power consumption is now different from original code and depends on how the compression utility works. Architectural side-effect attacks will be also private from direct interpretation of the results. This, of course, will bring a little extra effort (probably time), although the security expected is guaranteed by the AES algorithm.

In this work we have omitted some challenges in cryptography implementation. The first one is the key management. We simply assumed a model in which an asymmetric algorithm is used to establish a secure channel in order to exchange the public key used in AES. The second one is integrity verification. We don't deal with the problem in this paper, although we believe that compression can help in hashing as well.

Many works have proposed using bus encryption. Best was the first to propose the idea and holds some patents [11]. Gilmont *et al* [12] use prediction of fetches on a triple-DES and assumes the deciphering cost in about 2.5%. XOM [13] and AEGIS [14] are example of security platforms but the authors don't cope with deciphering overhead.

A recent paper from the AEGIS team faces the problem [10]. They decouple the AES computation from the fetch mechanism and do both in parallel. In such a case an OPT encryption, based on timestamps and AES is carried out. The retrieval of the original information is obtained, in the best case, with only a XOR operation in a block of 512 bits. The limitation in this approach is that an increase in main memory area is necessary, as the timestamps occupies space and should be kept along with the encrypted code. This is, probably, a problem for memory constrained embedded system. On the other hand, our solution is orthogonal to their approach, meaning that it can be used along.

Compression was already explored in [15] to minimize the self-reconfiguration of FPGAs, but in that approach a data compression algorithm can be used, as the bitstream is supposed to be completely decompressed before use. In [16] an overview of the bus encryption approaches is depicted, and a mention to the solution of using the CodePack [17] for the overhead problem is raised without any results.

6. CONCLUSIONS

This paper presents the study of using Code Compression to alleviate the bus encryption overhead problem in embedded systems. Our results show that by using a PDC code compression scheme we reduce the deciphering overhead in about 50% in average, for a set of Benchmarks from MediaBench and MiBench suites. Moreover the global performance of the system presents a speedup of 1.2 over the original code execution.

7. REFERENCES

[1] A. Wolfe and A. Chanin, "Executing compressed programs on an embedded RISC architecture," in *Proc. Int'l Symp. on Microarchitecture*, Nov. 1992, pp. 81–91.

[2] H. Lekatsas, J. Henkel, and W. Wolf, "Code compression for low power embedded system design," in *Proc. ACM/IEEE Design Automation Conference*, 2000, pp. 294–299.

[3] L. Benini, A. Macii, and A. Nannarelli, "Code compression for cache energy minimization in embedded systems," *IEE Proceedings on Computers and Digital Techniques*, vol. 149, no. 4, pp. 157–163, July 2002.

[4] E. W. Netto, R. Azevedo, P. Centoducatte, and G. Araujo, "Multi-profile based code compression," in *Proc. ACM/IEEE Design Automation Conference*, June 2004, pp. 244–249.

[5] E. Billo, R. Azevedo, G. Araujo, P. Centoducatte, and E. W. Netto, "Design of a decompressor engine on a sparc processor," in *18th Symposium on Integrated Circuits and Systems Design (SBCCI'2005)*, september 2005.

[6] N. I. of Science and Tecnology, "FIPS PUB 197: Advanced Encryption Standard," in *FIPS*. Natiional Institute of Science and Tecnology, 2001.

[7] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia communication systems," in *Proc. Int'l Symp. on Microarchitecture*, Dec. 1997, pp. 330–337.

[8] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, and T. Mudge, "Mibench: a free, commercially representative embedded benchmark suite," in *Proc. of the IEEE 4th Annual Workshop on Workload Characterization*, Dec. 2001, pp. 3–14.

[9] G. Gaisler, "Leon," [OnLine], Oct. 2003, available: http://www.gaisler.com.

[10] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Efficient memory integrity and verification and encryption for secure processor," in *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture. MICRO-36.*, 2003, pp. 339–350.

[11] R. M. Best, "Crypto microprocessor that executes enciphered programs," in *United State Patent 4,465,901*, 1984.

[12] J. J. Q. Tanguy Gilmont, Jean-Didier Legat, "Enhancing security in the memory management unit," in *Proceedings of the 25th EUROMICRO Conference*, vol. 1, 1999, pp. 449–456.

[13] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. B. end John Mitchell, and M. Horowitz, "Architectural support for programming languages and operating systems archive," in *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems table of contents - ASPLOS IX*, 2000, pp. 168–177.

[14] G. E. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas, "Aegis: architecture for tamper-evident and tamper-resistant processing," in *Proceedings of the 17th annual international conference on Supercomputing*, 2003, pp. 160–171.

[15] M. Huebner, M. Ullmann, F. Weissel, and J. Becker, "Real-time configuration code decompression for dynamic fpga self-reconfiguration," in *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004, p. 138b.

[16] R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, C. Anguille, M. Bardouillet, C. Buatois, and J. B. Rigaud, "Hardware engines for bus encryption: A survey of existing techniques," in *Proceedings of the conference on Design, Automation and Test in Europe*, vol. 3, 2005, pp. 40–5.

[17] M. Game and A. Booker, *CodePack: Code Compression for PowerPC Processors.* International Business Machines (IBM) Corporation, 1998.