Synergistic Integration of Code Encryption and Compression in Embedded Systems

Kamran Rahmani, Hadi Hajimiri, Kartik Shrivastava, Prabhat Mishra
Department of Computer & Information Science & Engineering
University of Florida, Gainesville, Florida, US
{kamran, hadi, kshrivas, prabhat}@cise.ufl.com

ABSTRACT

Code encryption is a promising approach that encrypts the application binary to protect it from reverse engineering and tampering, and decrypts the instructions during runtime. A major challenge is to trade-off between the security level and runtime decryption overhead. In this paper, we explore a synergistic combination of various code compression algorithms with code encryption techniques to reduce this overhead. Since decryption overhead (time) is linearly dependent on code size, it is promising to employ compression to reduce code size, and thereby achieve the advantages of both compression and encryption. Experimental results demonstrate that our proposed scheme can employ efficient encryption techniques while significantly improve the performance up to 2.3X (1.5X on average) and reduce energy consumption up to 57% (26% on average), compared to using encryption alone.

Categories and Subject Descriptors

C.3 [SPECIAL-PURPOSE AND APPLICATION-BASED SYSTEMS]: Real-time and embedded systems

General Terms
Design, Performance, Security

Keywords
Embedded systems, code compression, code encryption, energy optimization, performance optimization

1. INTRODUCTION

Embedded systems are used everywhere - starting from everyday appliances to complex safety-critical systems. In many scenarios, it is becoming exceedingly essential to keep these devices authentic and confidential. Encryption is widely used as a reliable way of protecting critical data for storage and transmission. For example, it is useful to encrypt network messages while sending them out through communication media and to decrypt them at the receiving end. Likewise, ciphering files while storing them on the hard disk protects them from being read in case the hardware itself is compromised. The process of encrypting binary code is different from that of other static data. Encrypting static data is mainly concerned with the complexity of the ciphering algorithm and the mode of operation. Binaries themselves can be encrypted as static data in the secondary storage and then decrypted while loading them in the main memory. For security reasons, it may be required to keep encrypted binary code in the main memory. This is required when the bus between main memory and secondary storage is not secure. In this situation, repeated fetching and decryption of blocks of code is going to produce an immense overhead which would render the execution extremely slow and infeasible in many scenarios.

Figure 1 shows the framework in which encryption and decryption are performed on application binaries in an embedded system. There are two stages of processing. During the offline stage, the binary code of the embedded system is first encrypted. Next, when the program is loaded, the code passes through the insecure channel between sender and memory and it is decrypted inside the embedded system. The encrypted code is stored in the primary memory and accessed by the processor. The decryption is done online (for each fetch). The time it takes to decrypt the code affects the performance of the system. Reducing the code size with code compression can reduce this decryption overhead significantly.

Figure 2 shows how code compression and associated decompression are performed in embedded systems. There are two stages similar to Figure 1. The first stage is offline, in which the code is compressed. The code is decompressed between the main memory and the processor to increase the effective memory size as well as to improve the performance.
3. CODE ENCRYPTION & COMPRESSION

Runtime decryption of encrypted code significantly increases instruction fetch delay. The main challenge is how to combine encryption with compression to keep performance as high as possible while maintaining the security of the system. As the decryption time is proportional to the size of code, combining encryption with compression is promising to improve the overall performance. However, combining both encryption and compression may lead to a number of problems. The main problem is that both decryption and decompression are slow and hence may prevent the full utilization of the processor performance. In order to get the best possible processor utilization, the decompression unit should be such that the rate at which instructions are fetched is equal to the rate at which the instructions are decompressed. This section describes the challenges associated with integration of encryption and compression and presents mechanisms to address these challenges.

3.1 Encryption followed by Compression

There are two ways of combining encryption with compression. The first scenario is shown in Figure 3. In this combination, the code encryption is followed by compression. The problem is that most compression algorithms take advantage of the repeating patterns in the uncompressed data set. Encrypted data generally has high entropy and therefore, has less similarity in patterns. As a result, as our experimental results show, it is difficult to compress those data.

3.2 Compression followed by Encryption

This is the most useful way of combining encryption and compression. In this combination the code compression is followed by encryption. It is beneficial to compress the unencrypted code by exploiting the regular pattern. Moreover, this compressed data can be easily encrypted and sent across the insecure channel to the receiving end. The decryption and the decompression units can do the rest of the work. This scenario is shown in Figure 4.

3.3 Placement of Cache

This section describes the challenges and opportunities associated with cache placement.
4 gives the ratio of cycles of encrypted and compressed code over regular code ($S_{N1}$) and Equation 5 gives the ratio for encrypted and compressed code over encrypted only code ($S_{E1}$).

$$S_{N1} = \frac{T_{ec1}}{T_{n1}} = C \left(1 + \frac{E_b + R_b}{M_b}\right)$$ (4)

$$S_{E1} = \frac{T_{ec1}}{T_{e1}} = C \left(1 + \frac{R_b}{M_b + E_b}\right)$$ (5)

The goal is to make $S_{N1}$ and $S_{E1}$ as low as possible. The obvious way to do so is to have a lower compression ratio $C$ and a low decryption latency $R_b$. We need to make a trade-off between these two factors. For example, Huffman coding gives a great compression but its decompression is slow. On the other hand, simple dictionary based compression gives low/moderate compression with faster decompression. Bitmap-based compression [18] provides a trade-off between these two aspects by providing good compression ratio with fast decompression.

### 3.3.2 PDCD Architecture

We can gain more benefit of code compression by storing compressed code in the cache. This can be done by putting the decompression unit between processor and cache. Figure 6 shows this architecture. In this scheme the encrypted code is fetched as blocks from the memory by the decryption unit, which are then decrypted and sent back to the cache. We refer this architecture as Processor-Decompressor-Cache-Decryptor (PDCD) architecture.

$$T_{n2} = N_b.M_b$$ (6)

$$T_{e2} = N_b.(M_b + E_b)$$ (7)

$$T_{ec2} = C.(N_b.(M_b + E_b) + N.R)$$ (8)
Like PCDD, the aim is to minimize $S_{N2}$ and $S_{E2}$. Choosing compression algorithm should be a trade-off between compression ratio and decompression unit speed. On the other hand, encryption algorithm should be chosen such that $S_{E2}$ is least. More secure algorithms need more cycles to decrypt. Hence, choice of an encryption algorithm leads to a trade-off between needed security and expected speed that depends on embedded system requirements. For example, AES will have larger decryption latency than DES. Hence $S_{E2}$ would be larger for DES and $S_{N2}$ would be larger for AES, i.e., execution of the encrypted and compressed code will be slower for AES compared to DES. Interestingly, the effect of compression would be more significant for AES as compression will hide more latency.

4. EXPERIMENTS

4.1 Experimental Setup

In order to explore different combination of code encryption and compression tradeoffs, we examined cjpeg, djpeg, epic, adpcm, rawcaudio and rawdaudio, g721 (encode, decode) benchmarks from the Mediabench [3] and dijkstra, patricia, crc32 from Mibench [12] compiled for the Alpha target architecture. All applications were executed with the default input sets provided with the benchmarks suites. Since the space is limited, we present the result for five of these benchmarks. However, the result is consistent for the remaining benchmarks.

Code encryption and compression are performed offline. In order to extract the code (instruction) part from executable binaries we used ECOFF header files provided in SimpleScalar toolset [9]. The text segment is extracted from the binary and compression is performed on it, giving compressed text segment as a result. Since the decompression unit must be able to start execution from any of the jump targets, branch targets should be aligned in the compressed code. In addition, the mapping of old addresses (in the original uncompressed code) to new addresses (in the compressed code) is kept in a jump table. This compressed text is then encrypted and a new binary file is created using the compressed-encrypted text, the dictionary, the jump-mapping table and the rest of the segments from the original file.

Three different code compression techniques including bitmask-based, dictionary-based and Huffman compression algorithms were used. To attain the best achievable compression ratios, in compression algorithms, for each application we examined dictionaries of 1 KB, 2KB, 4KB, and 8 KB. In addition, for bitmask-based compression similar to Seong et al. [18] we tried three mask sets including one 2-bit sliding, 1-bit sliding and 2-bit fixed, and 1-bit sliding and 2-bit fixed masks. We found that dictionary size of 2KB is the best choice for this set of benchmarks. Also, we examined compression word sizes of 8 bits, 16 bits, and 32 bits. We found out that 16 bits word size is the best choice for dictionary-based and Huffman compression algorithms. We used AES (128 bits block), DES (64 bits block), and RC4 encryption algorithms to examine the effect of compression on different classes of encryption methods (from strong-slow algorithm to weak-fast one).

To obtain cache hit and miss statistics, we modified the SimpleScalar toolset to be able to decrypt, decompress, and simulate encrypted-compressed applications based on PDCD architecture. Decompression unit can decompress the next instruction by one cycle (in pipelined mode) if it finds the entire needed bits in its buffer. Otherwise, it takes one cycle (or more cycles, if cache miss occurs) to fetch the needed bits into its buffer and one more cycle to decompress the next instruction. In decryptor, we used 18 [2], 11 [13], and 7 [14] cycles per byte latencies for AES, DES, and RC4 algorithms, respectively. Correctness of the compression and encryption algorithms was verified by comparing the outputs of encrypted-compressed applications with regular versions.

We applied the same energy model used in [20], which calculates both dynamic and static energy consumption, memory latency, CPU stall energy, and main memory fetch energy. The energy model includes decompression and decryption overhead energy. We used a single 1KB instruction direct cache with a line size of 16 bytes for all simulations. We refer it as base cache. We updated the dynamic energy consumption for this cache configuration using CACTI 4.2 [8].

4.2 Performance Improvement

Figure 7 shows the performance of applications in different combination of AES and compression algorithms normalized to the AES encryption only method. It confirms that code compression can improve performance in many scenarios while used with AES encryption algorithm. As we can see, performance improvement varies significantly and depends on the application binary. For instance, in the case of application g721_enc, applying compression would result in 1.1X, 1.2X, and 1.4X performance improvements for dictionary-based, Huffman, and bitmask-based algorithms, respectively. This improvement is up to 2.3X for bitmask-based algorithm in cjpeg application. On the other hand, in rawdaudio application we do not see any noticeable improvement. The improvement is significant if application code size and its behavior is such that it needs larger cache size than base cache (like cjpeg), and is negligible if application fits in the base cache effectively (like rawdaudio). The average performance improvements are 1.2X, 1.2X, and 1.5X for dictionary-based, Huffman, and bitmask-based algorithms, respectively. For ease of comparison, original numbers are shown in Table 1.

![Figure 7: Performance of different compression algorithms with AES encryption (normalized to AES without compression)](attachment:image)
Figure 8 illustrates the performance of applications for different combinations of DES and compression algorithms normalized to the DES encryption only method. As behavior of applications remains same for different encryptions, we see improvement pattern similar to AES case. For instance in the case of application g721_enc, similar to AES, applying compression would result in 1.1X, 1.2X, and 1.4X performance improvements for dictionary-based, Huffman, and bitmask-based algorithms, respectively. This improvement is up to 2.2X in cjpeg application. The average performance improvements are 1.2X, 1.2X, and 1.4X for dictionary-based, Huffman, and bitmask-based algorithms, respectively.

Performance improvements for different combinations of RC4 and compression algorithms is shown in Figure 9. As we discussed earlier, performance improvement is less in the case of faster decryption unit. We see this happens in RC4 that is faster and less secure than AES. For instance, in cjpeg we have 2.1X improvement in performance for RC4 with bitmask-based compression that is 2.3X for corresponding AES case. As cache misses are same in both cases, the improvement is larger for longer decryption latency. The average performance improvements are 1.2X, 1.2X, and 1.4X for dictionary-based, Huffman, and bitmask-based algorithms, respectively.

Figure 9: Performance of different compression algorithms with RC4 encryption (normalized to RC4 without compression)

On average we get most improvement in performance when we use bitmask-based compression with encryption algorithms. For this set of benchmarks, application code size is reduced by 15%-25%, 30%-35%, and 30%-45% for dictionary-based, Huffman, and bitmask-based compression, respectively. Bitmask-based compression is the best choice in terms of compression for this set of applications. The reason is that because of large similarity in instructions (that lets us use masks) we can use large 32 bits words and reduce the code size even more than Huffman algorithm (with restricted dictionary size).

The decompression hardware for dictionary-based compression is simple but average improvement is small. Bitmask-based compression is the best choice to be combined with all the three encryption algorithms in terms of performance improvement. This can result in up to 2.3X (1.5X on average) improvement in performance. This improvement can satisfy real-time requirements in many embedded applications while keeping them safe by using encryption methods.

### 4.3 Energy Savings

Energy consumption in instruction cache subsystem for different combinations of AES and compression algorithms is shown in Figure 10. As compression reduces the miss ratio in cache, it reduces the power consumption of the system. For instance in the case of patricia application, we have reduction of energy by 13%, 20%, and 24% for AES combined with dictionary-based, Huffman and bit-mask based compression, respectively. Energy saving can be even more significant when cache size is a bottleneck in the application. For example, we can save up to 57% of the total energy in the cjpeg application by combining bitmask-based compression with AES encryption. The average energy savings are 13%, 17%, and 26% for dictionary-based, Huffman, and bitmask-based algorithms, respectively. For ease of comparison, original numbers are shown in Table 2.

Figure 10: Energy consumption of different compression algorithms with AES encryption (normalized to AES without compression)

We have similar savings in other algorithms. Figure 11 illustrates energy consumption for different combinations of DES and compression algorithms. The average energy savings are 11%, 16%, and 26% for dictionary-based, Huffman, and bitmask-based algorithms, respectively.
Energy consumption for different combinations of RC4 and compression algorithms is shown in Figure 12. Like performance improvement, because of less latency in RC4 compared to AES, we have less energy saving in RC4 compared to AES. For instance in epicencode we have 20% energy saving for RC4 with bitmask-based compression whereas, it was 25% for corresponding AES case. On average, energy savings are 12%, 16%, and 25% for dictionary-based, Huffman, and bitmask-based algorithms, respectively.

In summary, like performance improvement, integration of compression with encryption would be useful in terms of energy consumption when the application needs larger instruction cache. As simulation results show, bitmask-based compression is the best choice in terms of both performance improvement and energy saving. On average, by using this algorithm with AES encryption we can save 26% of total energy and improve the performance by 47%.

5. CONCLUSIONS

Encryption and compression are important for embedded systems. While the former provides code security and prevent tampering by third party, the latter is used to minimize the code size and thus reduce power and memory requirements as well as improve the overall performance. In this paper, we have demonstrated that it is useful to first compress the code and then encrypt it, employing a Processor-Decompressor-Cache-Decryptor architecture. Since code size is reduced due to compression, the decryptor has to operate on less amount of code, which makes it faster. Our experimental results demonstrated up to 2.3X (1.5X on average) improvement in performance and up to 57% (26% on average) energy saving by combining compression and encryption compared to employing encryption alone. This improvement can enable use of encryption in embedded systems.

6. ACKNOWLEDGMENTS

This work was partially supported by NSF grant CNS-0915376. We would like to thank Kanad Basu and Yogesh Sharma for their comments and suggestions.

7. REFERENCES