Using Value Locality to Reduce Memory Encryption Overhead in Embedded Processors

G. Keramidas, P. Petoumenos, A. Antonopoulos, S. Kaxiras, D.N. Serpanos
Department of Electrical and Computer Engineering
University of Patras
GR-26504 Patras, Greece
{keramidas,ppetoumenos,antonopa,kaxiras,serpanos}@ece.upatras.gr

Abstract

Memory encryption has gained much attention lately as a way to offer a secure environment to fight against software and hardware attacks. Many researchers provided memory encryption schemes whereby one or more levels of the memory hierarchy were encrypted using a cryptographic algorithm such as AES. Counter Mode (CM) encryption, also called One-Time-Pad (OTP) encryption, is proven to be quite effective for main memory encryption. However, CM encryption requires an extra sequence number (counter) to be associated with every memory location (L2 block cacheline granularity is used). The per-block counters must be updated every time a block is written back to memory otherwise known-plaintext attacks may occur. Thus, the size of those counters is a critical parameter in the system design. In this work, we propose the use of silent stores as a method of providing the CM encryption with less overhead. Silent stores, i.e. stores, to memory that write the same value as already stored in that memory location, have been observed to occur frequently. These stores create redundant memory write-backs (and counter updates), so eliminating them will lower performance overheads introduced by the en-cyption/decryption process. Our initial results show significant benefits across the board indicating the promising nature of the proposed idea.

1. Introduction

There is a growing interest in creating processing environments resistant to software piracy and security attacks. While most of the attention has been given to (usually remote) software-only attacks, hardware attacks are receiving much attention too. One characteristic example is the X-BOX game console where unauthorized playing of games can be accomplished by inserting an inexpensive chip on the bus [3]. In the past, many software companies fell victim to such violations —usually called software piracy— resulting to a damage of millions of dollars. Therefore, it can be expected that security issues will join performance and power issues in building future computer systems

One critical design component of these security systems is memory encryption that provides protection for program and data privacy. For such security systems, memory encryption often lies at the centre of protection. It is primarily used to create the necessary framework to avoid malicious attacks i.e. code injection attacks. Due to this increasing interest in creating security platforms, many computer architects have turned their attention in providing security enhancements in the memory subsystem components in uniprocessor [1][2][4][5][6][7][17] as well as in multiprocessor systems [8][9].

The execute-only-memory model (XOM), was one of the initial attempts to support copy and tamper resistant software communication [1]. Most of the recent work in secure computing architecture is inspired by XOM. XOM drew a perimeter of trust around the CPU and grew trust out from there. In XOM, application programs and their data are stored encrypted in the main memory (insecure area) while the on-chip caches are considered secure (data and instructions are stored in a plaintext format). Figure 1 illustrates the abstract model of the crypto-procedure (assuming a two-level cache structure). Writing to memory

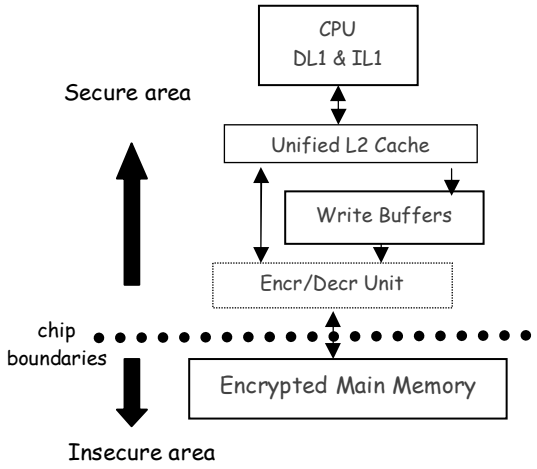

Figure 1. The XOM architecture

is deferred through the write buffer. Every dirty L2 cache line is encrypted first and then send down to memory. Likewise, every line read from the main memory is decrypted before it is stored in the L2 cache and used by the CPU.

Despite its novelty, the XOM architecture was not designed having performance as a prime consideration. Memory encryption/decryption lies on the memory access critical path and therefore the performance decrease is obvious. Every memory transaction experiences the encryption/decryption latency which may take-up to hundred of cycles using the popular AES algorithm [10][11]. As observed by other researchers, the performance overhead of using memory encryption in this way is very high, slowing down the program execution up to 35% even when a very simple encryption algorithm was used.

To remedy the latency problem, a different mode of AES encryption mode is used: one time pad (OTP), also called Counter Mode (CM) [4][5]. The next section presents how the OTP mode encryption scheme shifts the computational intensive crypto-process off the critical path and introduces some new problems related with this mode of encryption.

2. OTP Encryption

OTP mode encryption was proposed as an alternative solution to circumvent the additional AES latency introduced by the sequential nature of the direct encryption mode [12]. Instead of applying AES directly to data or code, CM encryption applies AES to a seed to generate a pad (Figure 2). Data is then encrypted and decrypted via a simple bitwise XOR with the pad. The key issue is that the fetch of an encrypted block from the main —encrypted— memory and the pad computation of that block can be overlapped hiding much of the computational intensive crypto-process. Assuming that the latency of the decryption is less than the latency of a load, the only additional latency on the critical path that decryption introduces is the latency of the XOR operation, which is negligible. However, CM relies on the fundamental assumption that the same seed will never be used more than once with a given AES key [12]. This is because the same seed and AES key will produce the same pad and as a result it is a straightforward process to reproduce the plaintext of a memory block if the plaintext of another block (or a previous value of the same block) is known or can be guessed (this kind of attack is called known plaintext attack).

Two approaches are used in order to ensure the uniqueness of the seeds for memory encryption. According to the first one, a single global counter is used as a seed for all memory blocks [4][5]. This counter (also called

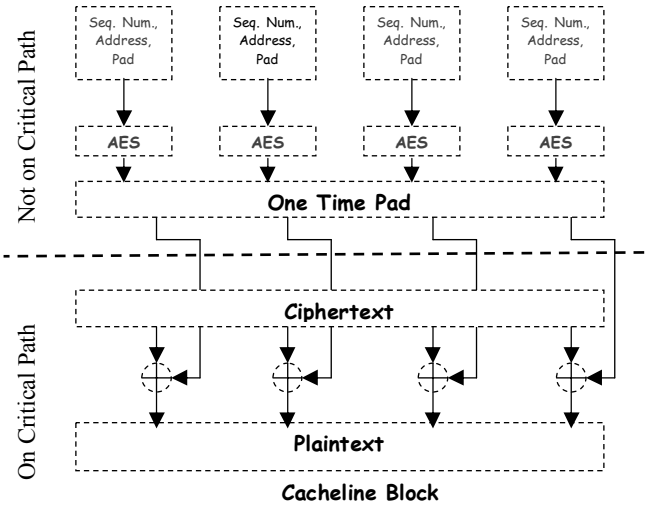

Figure 2: DTP Decryption

sequence number) is incremented every time a memory block write-back happens. The second approach utilizes the idea of per-block (local) counters. In this case the unique seed is formed as a concentration of the blocks address and its local counter. The address part assures that different seeds are used for the encryption of different memory locations. The local counters attached to each block are incremented on each write-back of that block, so every time a write-back occurs, a different seed is used for encryption.

The length of the counters is a critical parameter for the system design since it has to be short enough to store compactly but long enough to prevent frequent wrap-around. Every time a counter overflows, the AES encryption key is changed to prevent re-use of the same pad. But since the same encryption key is used to enctypt the whole memory, a key change dictates re-encyption of the entire memory. Such a re-encryption process can stall the system for a conspicuous time rendering the above scenario a cumbersome solution for todays high performance systems. In [17], it is mentioned that re-encryption of 4 GB of memory at a rate of 6.4 GB/second freezes the system for nearly one second. Using small counters, re-encryptions will happen very frequently, however increasing the size of the counters affects not only the amount of memory used to keep them, but more importantly the performance of the memory encryption process.

In OTP encryption, the value of the counter (sequence number) used to encrypt a block is also needed to decrypt it. Thus, the counter value must be immediate available to the pad generation hardware, otherwise the hiding of the decryption latency will not be achieved. Since every memory block —considering a L2 block size granularity— has its own local, it is not possible to devote on-chip

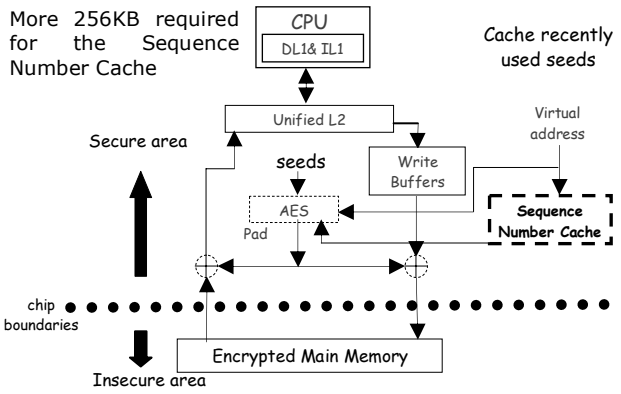

Figure 3: Design of OTP Encryption on data with sequence number chache

memory to store them. The simplest solution is to cache recently used per-block counters [4][5]. To avoid polluting the common on-chip caches with per-block counters, a special cache, called Sequence Number Cache (SNC), may be employed (Figure 3). While this eliminates decryption latency in many cases, misses in the SNC result in added latency because the per-block counter must be fetched from main memory before pad generation can begin. To be effective, the SNC must be sufficiently large. Previous work proved that more than 256KB of on-chip storage must be devoted to the SNC in order to report sufficient hit rates [4][5][7].

Two recent approaches were proposed to optimize the performance of the OTP encryption. Shi et al. [7] propose a series of techniques to predict sequence numbers and pre-compute OTP pads so that the correct OTP pad is available by the time the ciphertext data comes in from the main memory. In [17], the authors introduce a hierarchical counter scheme, using per-block local counters and per-virtual page global counters, to limit the counter overflow events as well as to increase the efficiency of the SNC (since smaller counters can be used).

Both approaches, although very successful, try to attack the consequences of the frequent counter updates. In this work, we move one step behind; our solution is based on a special form of value locality, called temporally silent stores. The next section explains the silent store property, presents some initial results, and discusses how the proposed idea can be used to increase the performance of the OTP encryption in typical embedded and networking applications.

3. Exploiting Value Properties To Reduce Memory Encryption Overhead

Value locality and value prediction [13] has been presented as a very promising technique to maximize the number of on-chip operations as opposed to climbing the "memory wall" and suffering the lower speeds of buses and main memory. One kind of value locality, called silent stores, happens when a store instruction writes a value matching the one already stored at the main memory location. Previous work has shown that silent stores occur quite frequently and can be exploited to reduce memory traffic and improve performance [14]. The idea of silent stores was further explored to encompass a new set of stores that change the value stored at a memory location, but only temporally. For example, if a value is altered and promptly returns to the same value in a subsequent write, it is still marked dirty and is written back to main memory even though the value in main memory is identical. While each store is not silent (does not attempt to write the same value), the cumulative effect of the stores is silent. This property is called temporal silence, and it has been proved to be quite frequent in scientific, commercial, networking and embedded benchmarks [15][16].

3.1 Indentifying and Squashing Temporally Silent Stores

In this subsection, we outline various ways for detecting temporally silent stores, presented in previous work, and we point out which method, called dusty cache, is appropriate in our case. Finally, we present our ideas to optimize the performance of the dusty cache approach.

The methods for identifying temporally silent stores fall into two categories: i) methods inside the processor core, and ii) methods outside the processor core. The first category typically includes the utilization of the load/ store queue (LSQ). This is a straightforward procedure. Detecting temporal silence simply involves checking any new store data values against system visible value for that location (the first loaded value for the location of interest in the LSQ). Any stores which become temporally silent in the queue can be dropped, while all other stores are performed as usual. Although this method has been proven to be successful in detecting normal silent stores [14], its ability in detecting temporally silent stores is quite low [15] (probably because the distance between the intermediate value store and the temporally silent store is large enough to be captured by the relatively small buffering capabilities of the LSQ).

Detecting temporal silence outside the processor core can be achieved either by exploiting the natural behaviour of inclusive cache hierarchies or by augmenting one or more levels of the memory hierarchy with additional data storage to keep the initial data values of the cached data (the dusty cache approach). According to the first method, when the CPU writes a cache-line (assuming a write-back Data L1 with an inclusive L2), the line is inserted into the DL1 and it is written while the L2 updates its dictionary to indicate that the line was modified. Since the L2 contains the initial data value, we can use it to detect temporally

silent stores. However this method is quite limited since it cannot be employed by the very common non-inclusive caches.

The dusty cache approach involves adding a second data cache bank that acts as an image of the main memory (image cache) in one or more levels in the memory hierarchy. When the CPU issues a new memory request, the requested cache-line is saved both in the normal cache and the image cache. Upon a write instruction, the new write value is both written into the normal cache and compared against the corresponding context of the image cache. If a match occurs, temporal silence for that write has been detected. In previous work, the image cache and the data cache array are of equal size. Our preliminary data indicates that only a few cache-lines and a few words per cache-line need to be tracked. However, we leave detailed exploration of this design space to future work.

3.2 Evaluating Temporally Silent Stores in Embedded and Networking Applications

The benchmark set we use for our evaluation targets typical embedded and networking applications. We use three applications from the MiBench [19] and four applications from the Commbench [18]. The MiBench suite is designed to be representative of a wide range of embedded applications and the CommBench suite is designed to be representative of networking applications. Table 1 shows the total number of loads and stores for each of the benchmark application (results taken from [16]).

Table 1. Total number of load and store instructions for each benchmark

Benchmarks	loads	stores
basicmath	74,151,136	46,577,732
bitcnts	68,009,977	15,626,007
dijkstra	60,985,966	8,925,084
drr	180,171,104	92,676,773
frag	178,058,884	96.099,734
reed enc	149,063,969	62,261,586
reed dec	208,593,061	89,553,034
sha	424,476,497	158,084,970

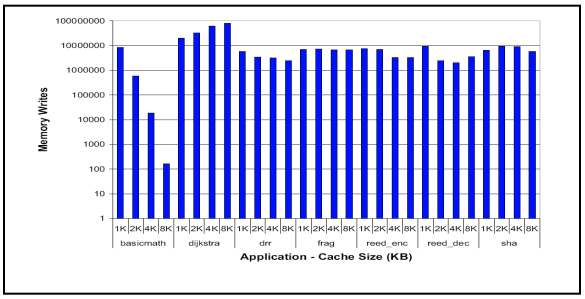

Figure 4: Memory write-backs appeared in each application

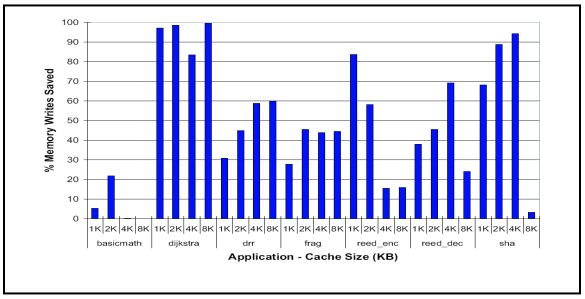

Figure 5: Percentage of memory writes saved with a dusty cache

Figure 4 shows, for each application and each cache size, the total count of memory writes that occur in the program execution. Given that this is a write-back cache, these writes to the memory subsystem occur primarily as a result of cache evictions. Figure 5 shows the savings in memory writes (as a percentage of the original number of memory writes shown in Figure 4) for each application and cache size when using a dusty cache. Note that for the moment, we do not take into account the extra on-chip memory required to capture the temporal silence functionality. This is part of the next subsection. The results shown in Figures 4 and 5 are a measure of the frequency of silent stores that potentially can be squashed, irrespective of the cost of the method.

As we can see from Figure 5, the savings are quite substantial, frequently well above 60%. What is noteworthy is the surprisingly large fraction of silent stores in the dijkstra benchmark (the percentage of the silent stores approaches the 100%). The above results indicate that the temporal silence property is quite frequent in embedded and networking applications. The next subsection discusses how this property can be used to increase the performance of the OTP encryption.

3.3 Reducing the OTP Encryption Overhead

As we have already mentioned, the OTP mode encryption relies on the fundamental assumption that the same seed will never be used more than once with a given AES key [12]. This is because unique sequence numbers are required in the encryption and decryption process, otherwise known plaintext attacks may occur. To ensure unique seeds for memory encryption, per-block local counters are used. These counters are incremented on each write-back to that block. In this way the same seed will never be used to encrypt the same memory location. However, the write-backs of a block could be frequent and counters quickly become large. Unfortunately, when a counter of one memory block overflows, the AES encryption key is changed to prevent the re-use of the same pad. The problem is that the same key is used to encrypt the entire memory, so a key change requires re-encryption of the entire memory. Such re-encryptions can stall the system for a noticeable time.

Detecting and squashing temporally silent stores will make the counters to count as slow as possible. Increasing the size of the counters is not a solution because this will degrade memory encryption performance. But reducing the triggering events (write-backs in our case) by eliminating the unnecessary (silent) write-backs, the frequency of system "freezes" is reduced. System "freezing" may be very catastrophic in real-time or in interactive applications and/or may arise opportunities for the attackers since during re-encryptions the system becomes vulnerable to attacks. Furthermore, an extra feature of the temporally silent stores makes them especially suitable in our case. As it is mentioned in [15], frequently updated memory locations are likely to be temporally silent. Since re-encryption of the main memory is required when a single local counter wraps-around, eliminating the frequently updated silent stores will further increase the effectiveness of the proposed idea.

From another point of view, for OTP encryption to be effective in eliminating the decryption latency, the per-block counters must be immediate available to the pad generation hardware. As we have already mentioned, since every memory block has its own local counter, it is not possible to devote on-chip memory to store them, thus a special cache, called Sequence Number Cache (SNC), is employed. The SNC must be effective because misses in the SNC result in added latency since the per-block counter must be fetched from the main memory before pad generation can start. Previous work proved that more than 256KB of on-chip storage must be devoted to the SNC in order to report sufficient hit rates [4][5][7].

Under this scenario, employing the temporal silence feature will allow us to decrease the size of the per-block counters (for a given frequency of system "freezes"). Using smaller counters affects not only the amount of off-chip memory used to keep them but more importantly it will lead to a better utilization of the on-chip storage. A given SNC hit rate can be achieved with less on-chip storage or alternatively the same transistor budget will produce higher SNC hit rates which is, as we have seen, a critical parameter for the performance of the OTP encryption.

However, indentifying and squashing temporally silent stores is not free of cost, especially when the dusty cache approach is used. Recall that the dusty cache approach involves adding a second data cache bank that acts as an image on the main memory in one or more levels of the memory hierarchy. Duplicating the data array of the Data L1 cache is not an issue in our case. As Figure 5 indicates, 1KB to 8KB dusty cache sizes are able to capture the bulk of the temporally silent stores appearing in the program execution. Prior work suggests to devote more that 256KB of on-chip storage for the SNC. On the other hand, if someone wants to employ the dusty cache approach to the relatively large L2 caches, the amount of memory that is required in this case is quite substantial. Our solution to this is that the system L2 cache and the dusty cache should not be of the equal size. Our preliminary results indicate that only a few cache-lines and a few words per cache-line need to tracked, while the most of the silent store functionality can still be captured. However, we leave detailed exploration of this design space to future work.

4. Conclusions

General-purpose secure processing has been receiving increasing attention in computer architecture. For such security systems, memory encryption often lies at the center of protection. In this paper we have looked at all recent proposals on how to built efficient memory encryption schemes and we highlight the advantages and the disadvantages of the proposed techniques. Finally, we present how the temporally silent store feature can be used to improve the performance of the memory encryption process in the high sensitive embedded processor domain. Our proposal is orthogonal to the previous proposed memory encryption architectures. Our preliminary results show significant benefits across the board indicating the promising nature of the proposed idea.

5. Acknowledgements

This work was co-funded from the European Union by 75% and from the Hellenic State - Greek Ministry of Development - General Secretariat of Research and Development (GSRT) by 25% through the Operational Programme Competitiveness, 2000-2006.

References

[1] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horwitz. Architectural Support for Copy and Tamper Resistant Software. International Conference on Architectural Support for Programming Languages and Operating Systems, 2000.

[2] R. B. Lee, P.C. S. Kwan, J. P. McGregor, J. Dwoskin, and Z. Wang. Architecture for protecting critical secrets in microprocessors. International Symposium on Computer Architecture, 2005.

[3] A. Huang. Hacking the Xbox: An Introduction to Reverse Engineering. No Starch Press, 2003.

[4] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processor. International. Symposium on Microarchitecture, 2003.

[5] J. Yang, Y. Zhang, and L. Gao. Fast Secure Processor for Inhibiting Software Piracy and Tampering. International Symposium on Microarchitecture, 2003.

[6] B. Gassend, G. Suh, D. Clarke, M. Dijk, and S. Devadas. Caches and Hash Trees for Efficient Memory Integrity Verification. International Symposium on High Performance Computer Architecture, 2003.

[7] W. Shi, H.-H. Lee, M. Ghosh, C. Lu, and A. Boldyreva. High efficiency counter mode security architecture via prediction and precomputation. International Symposium on Computer Architecture, 2005.

[8] W. Shi, H.-H. Lee, M. Ghosh, and C. Lu. Architectural Support for High Speed Protection of Memory Integrity and Confidentiality in Multiprocessor Systems. International Conference on Parallel Architectures and Compilation Techniques, 2004.

[9] Y. Zhang, L. Gao, J. Yang, X. Zhang, and R. Gupta. SENSS: Security Enhancement to Symmetric Shared Memory Multiprocessors. International Symposium on High-Performance Computer Architecture, 2005.

[10] P. R. Schaumont, H. Kuo, and I. M. Verbauwhede. Unlocking the design secrets of a 2.29 gb/s Rijndael processor. In Design Automation Conference, 2002.

[11] H. Kuo and I. M. Verbauwhede. Architectural optimization for a 1.82 gb/s VLSI implementation of the AES Rijndael algorithm. In Cryptographic Hardware and Embedded Systems, 2001.

[12] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption: Analysis of the des modes of operation. International Symposium on Foundations of Computer Science, 1997.

[13] M. H. Lipasti, C. B. Wilkerson, and J. P. Shen. Value Locality and Load Value Prediction. International Conference on Architectural Support for Programming Languages and Operating Systems, 1996.

[14] K. M. Lepak and M. H. Lipasti. Silent stores for free. International Symposium on Microarchitecture, 2000.

[15] K. M. Lepak and M. H. Lipasti. Temporally silent stores. International Conference on Architectural Support for Programming Languages and Operating Systems, 2002.

[16] P. Krishnamurthy, R. D. Chamberlain, and R. K. Cytron. Evaluating Dusty Caches on General Workloads. Workshop on Duplicating, Deconstructing, and Debunking, 2006.

[17] C. Yan, B. Rogest, D. Englender, Y. Solihin, M. Prvulovic. Improving Cost, Performance, and Security of Memory Encryption and Authentication. International Symposium on Computer Architecture, 2006.

[18] T. Wolf and M. A. Franklin. CommBench - A telecommunications benchmark for network processors. International Symposium on Performance Analysis of Systems and Software, 2000.

[19] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A free, commercially representative embedded benchmark suite. Workshop on Workload Characterization, 200