Memory Predecryption: Hiding the Latency Overhead of Memory Encryption

Brian Rogers, Yan Solihin
Dept. of Electrical and Computer Engineering
North Carolina State University
{bmrogers,solihin}@eos.ncsu.edu

Milos Prvulovic
College of Computing
Georgia Institute of Technology
milos@cc.gatech.edu

Abstract

Memory encryption has become a common approach to providing a secure processing environment, but current schemes suffer from extra performance and storage overheads. This paper presents predecryption *as a method of providing this security with less overhead by using well-known prefetching techniques to retrieve data from memory and perform decryption before it is needed by the processor. Our results, tested mostly on SPEC 2000 benchmarks, show that using our predecryption scheme can actually result in no increase in execution time despite an extra 128 cycle decryption latency per memory block access.*

1. Introduction

Increasingly numerous and sophisticated security threats are creating a growing need for processing environments resistant to software piracy and security attacks. While remote software-only attacks are receiving much of the attention in this area, hardware attacks are also possible and feasible, for example on the X-Box game console where a bus or DRAM override can be accomplished by inserting an inexpensive chip on the bus, allowing unauthorized playing of games [7]. Such violations of the Digital Rights Management (DRM) policy can be very costly - software piracy has cost the software industry billions of dollars per year in recent years due to unauthorized copying of restricted software [9]. One promising solution for hardware-based attacks is to encrypt data as it leaves the processor chip and decrypt it when it is brought back on-chip.

Recent studies in computer architecture have proposed a model referred to as the execute-only memory (XOM), as a way to support copy and tamper resistant software communication [6, 13, 14]. In XOM, application programs and their data are stored encrypted in the main memory. Unfortunately, as observed by other researchers, the performance overhead of using memory encryption in this way is very high, slowing down application programs by up to 35% even when using a very simple encryption algorithm that takes only 48 cycles to decrypt a cache line [21]. This is because each memory access suffers from decryption latency, which may take up to hundreds of cycles using the popular AES algorithm.

To lower performance overheads, recently proposed schemes rely on approximating one-time pad (OTP) encryption [20, 21] to allow overlap between a memory access and decryption. In these schemes, memory blocks are not directly encrypted using a strong key-based scheme. Instead, the key is used to encrypt a "unique identifier" (pad) of the block, obtained from the block's address and a sequence number. The resulting pad for the block is then XORed with the plaintext of the block to produce cyphertext. When the block is fetched from memory, it is decrypted by recomputing its pad and XORing it with the encrypted block. The fetch of an encrypted block from memory and the pad computation for that block can be overlapped, hiding much of the latency of encryption/decryption. However, this approach has a significant drawback: the pad must be different each time a block is encrypted [20, 21]. If a pad is reused and the attacker knows, discovers, or guesses the plaintext of one data block, it is a simple matter to recover the pad (by XORing the known plaintext and its cyphertext) and use it to decrypt all other blocks that used the same pad. In memory systems, many locations have known or easily guessable values (e.g. many are zeroes), so using the same pad more than once is very risky. Because the address of a data block remains constant, the uniqueness of the pad for different versions of that block relies on the sequence number used to generate the pad. Maintaining such sequence numbers is a significant problem. The sequence number is kept on a per-block basis, so the OTP-like scheme must record the current sequence number for each block in memory. Pad generation cannot begin until the block's sequence number is fetched, so most of the latency-hiding opportunity is lost unless sequence numbers are kept in fast and relatively large on-chip storage. On-chip storage is still not sufficient for sequence numbers of *all* blocks in memory, so storage of sequence numbers also imposes a significant memory space overhead. Finally, when sequence numbers wrap around, the encryption key must be changed to avoid using the same pads again. This requires re-encryption of the entire memory, which is a significant overhead. The frequency of wrap-arounds can be reduced by using large sequence numbers, but then even more space is needed to store them.

To avoid the complexities and disadvantages of OTP-approximating schemes, in this paper we evaluate the feasibility of using well-known latency-hiding prefetching techniques to minimize the overheads of memory encryption. We augment the processor with a prefetching engine that predicts future cache misses, prefetches, and decrypts them ahead of the processor's requests. We call this scheme *predecryption*. The prefetcher that we use includes stream buffers [3, 5, 8, 11, 15, 17] and a correlation prefetcher [1, 2, 10, 12, 17, 18]. Compared to XOM and OTP, this approach has several benefits. First, prefetching engines are already present in real systems [5, 8], so little extra hardware is needed, other than tuning it to also hide the decryption latency. Furthermore, more advanced predictors will not only hide decryption latency better, they may also improve performance by also hiding memory latency. Second, predecryption allows well-known encryption schemes, such as AES, to be directly used to encrypt data in memory. In contrast, OTP-like schemes hide decryption latency by modifying the encryption scheme. Third, unlike OTP-like schemes, predecryption does not use sequence numbers, so no memory space is needed for them and there is no need to periodically re-encrypt the entire memory when a sequence number wraps around. Finally, when the data needs to be communicated to other devices, such as other processors in a multiprocessor system, the key needs to be passed to these devices once. In OTP-like schemes, the sequence number needed to generate the pad for a block must be communicated between devices together with the encrypted block itself.

The paper is organized as follows: Section 2 discusses the predecryption algorithm used and compares it with OTP, Section 3 details the evaluation setup, Section 4 presents and discusses the evaluation results, and Section 5 summarizes our findings and conclusions.

2. Predecryption Mechanism

Figure 1 shows the mechanism for predecryption. When an L2 cache line is replaced or flushed, it is encrypted before it is written back to the memory (Step 1a). A writeback is typically not a latency-critical operation, therefore we do not attempt to hide the encryption latency. Instead, we mainly focus on hiding the decryption latency, which affects time-critical fetches from memory into the L2 cache upon an L2 cache miss. When an L2 cache miss occurs, instead of requesting the line from memory, we first check the predecryption buffer for a match (Step 1b). If the missed line is found in the predecryption buffer, it is moved to the L2 cache. Otherwise, the miss is forwarded to the memory controller for fetching and to the prefether (Step 2). The prefetcher's stream buffers use the miss to identify streams and predecrypt them. If the miss is not identified as a part of a stream, the miss is forwarded to the correlation prefetcher to record the address in its table and make its prediction on future misses. Thus, the correlation prefetcher only sees addresses that are not sequential. In this way, the correlation table can be smaller because it is only used for "difficult" misses that can not be handled by stream buffers. A similar optimization has been used in past studies on prefetching [17, 18].

When the prefetcher observes a miss, its stream buffers or its correlation prefetcher predict future miss(es) and issue a predecryption request to the memory controller (Step 3). When the main memory replies with data (Step 4), if it is a reply to a read/write miss, it is decrypted and inserted into the L2 cache (Step 5a). If it is a reply to a predecryption request, the data is decrypted and stored in the predecryption buffer.

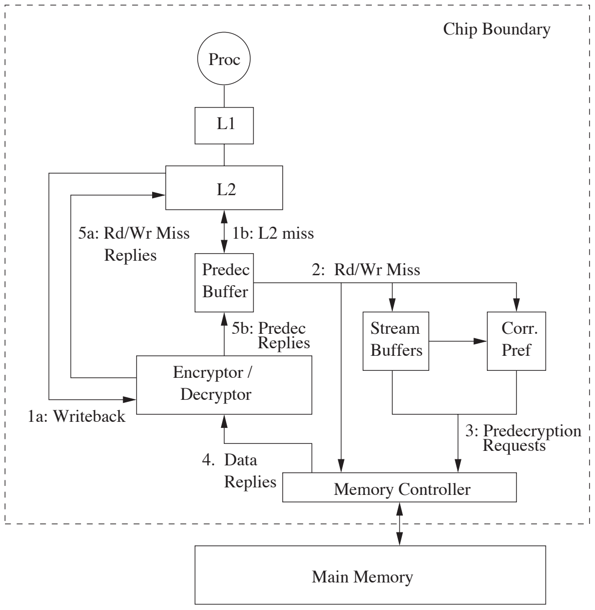

Figure 1: Predecryption mechanism for a processor with two levels of on-chip caches.

As long as the predecryption mechanism is able to predict far enough ahead, both the decryption latency and the memory latency can be hidden. Therefore, we tune the prefetcher to predict future misses far into the future. To achieve that, we use deep stream buffers, with eight entries per buffer, and other standard features such as a multiple-stream detection capability, double Δ scheme, and a heuristic to avoid storing overlapping streams [4]. For correlation prefetching, we use a replicated table organization that prefetches several levels of successors for the current miss address [18].

2.1. Qualitative Comparison with OTP

Table 1 qualitatively compares the properties of our predecryption scheme with encryption based on One-Time Pad (OTP) approximation, which was previously proposed [20, 21]. For a cache miss, in the best case scenario, OTP finds the needed sequence number on chip in the *sequence number cache* (SNC), and can overlap pad generation latency with memory latency. The best case for predecryption is when the requested line has already been fetched and decrypted into the predecryption buffer, in which case predecryption completely hides decryption and memory latencies. The worst-case latency for OTP is when the sequence number is not found in the SNC and needs to be fetched before decryption can be performed, in which case the sequence number must be retrieved from memory before pad generation can begin. Therefore, the overall latency of the entire operation is the sum of memory and pad generation (encryption) latency. The worst-case latency for predecryption is the same, because a cache line must be fetched and then decrypted before it can be used by the processor.

In terms of storage, OTP stores a sequence number for each line, in the SNC on-chip, or off-chip. Predecryption requires prediction information on-chip. If only stream buffers are used, they occupy little storage space, in our case 512 bytes to store information from multiple streams and 16KB for their predecryption buffer. However, to perform well a correlation table may need to be very large, such as 1 MB used in past studies [18, 1, 2, 10, 12]. Reducing the size of the correlation table without significantly affecting its effectiveness is a topic for future work.

OTP also complicates communication with other processors in a multiprocessor environment and with other devices. When an off-chip device accesses a datum, it needs the encryption key and the sequence number. The sequence number is different for each line and must be fetched from memory or, if the number is still in the SNC, from the processor that last updated the line. This could be a significant overhead, especially for parallel or I/O intensive programs. In contrast, with predecryption external devices only need the key, which can be supplied to them once. Therefore, predecryption has virtually no additional run-time overhead for parallel and I/O intensive programs.

Finally, it should be noted that OTP cache misses that do not find their sequence number in the SNC cannot directly benefit from prefetching. Even if the cache miss latency is hidden by prefetching, the sequence number of the line still has to be fetched. Therefore, unless there is a mechanism for predicting the next sequence number that is used, OTP cannot benefit much from prefetching. However, we will look into this issue in the future to possibly combine the advantages of both schemes.

Overall, in terms of best-case latency, on-chip and off-chip storage overheads, and complexity, predecryption is a very promising alternative to OTP-approximation encryption schemes.

3. Evaluation Setup

To evaluate our approach, we use 18 applications, mostly from Spec2000 [19]. Table 2 lists these applications and their characteristics. We perform detailed execution-driven simulation of a system whose relevant parameters are shown in Table 3, together with descriptions of specific configurations used in our experimental evaluation.

The correlation table has 64K entries. Each entry contains a tag of 11 bits, and multiple successors, where each successor is stored as an offset to the current miss address, and is encoded with 17 bits. Therefore, the total table size is $(64K \times (11 + 4 \times 17))/8 = 647$Kbytes.

4. Evaluation

Figure 2 plots the execution time for each benchmark and the average of all the benchmarks, when no encryption is applied (*NoEnc*), when encryption is applied (*Enc*), and when various predecryption schemes are applied (*Enc+Sbuff*, *Enc+CP*, and *Enc+CP+Sbuff*). All bars are normalized to the *NoEnc* case. The figure shows that on average, adding a 128 cycle encryption/decryption delay to all memory accesses increases the execution time by 21%. Predecryption with stream buffers (*Enc+Sbuff*) reduces this overhead to a little over 1%, whereas a correlation predictor alone (*Enc+CP*) reduces this overhead to 16%. A combination of both predecryption schemes (*Enc+CP+Sbuff*) results in the same execution time as in the *NoEnc* configuration. This result shows that, as explained in Section 1, a predecryptor eliminates some of the performance lost to memory encryption. However, Figure 2 does not indicate what part of the miss latency is hidden by predecryption: memory latency alone, or both memory and decryption latency. Figure 3 helps answer this question.

Figure 3 shows the execution time reduction due to prefetching/predecryption in a system with or without memory encryption. The first pair of bars in each group shows the relative reduction in execution time

Aspect	OTP-Approximation Approach	Predecryption Approach
Best-case latency	Cache *miss latency*	Predecryption buffer *hit latency*
Worst-case latency	Cache miss + decryption latency	Cache miss + decryption latency
On-chip storage overhead	Sequence Number Cache	Stream buffers + correlation table
Off-chip storage overhead	Sequence number storage	None
Other overheads	Re-encrypt entire memory on seq. num. wraparound	None
Communication with other procs	Transfer sequence number on each instance + key once	Transfer key once

Table 1: Qualitative comparison between our predecryption approach and OTP-approximation approach.

Benchmark	Source	Input Set	L2 Cache Global Miss Rate	L2 Cache Local Miss Rate	Dynamic Instructions
applu	Spec2K	train	2.96%	81.71%	15,559 M
bt	NAS	class A	0.09%	4.37%	560 M
bzip2	Spec2K	train	0.15%	8.02%	6,696 M
cg	NAS	class S	1.88%	18.90%	289 M
equake	Spec2K	train	2.87%	78.90%	25,120 M
euler	NASA	euler.in	0.51%	2.4%	2,793 M
ft	NAS	class S	1.91%	15.86%	1,567 M
gap	Spec2K	train	2.72%	55.42%	6,733 M
irr	PDE solver	10K nodes and 1M edges	1.52%	3.41%	963 M
is	NAS	class A	5.45%	35.55%	3,373 M
lu	NAS	class A	3.49%	55.19%	2,844 M
mcf	Spec2K	train	7.16%	26.39%	6,398 M
mgrid	Spec2K	$64 \times 64 \times 64$ grid, 3 iters	1.41%	55.05%	28,603 M
moldyn	Molecular dynamics code	moldyn.in	0.16%	2.63%	4,954 M
mst	Olden	1024 nodes	1.36%	37.34%	517 M
parser	Spec2K	train	0.39%	9.77%	7,811 M
sp	NAS	class A	2.63%	46.01%	2,825 M
swim	Spec2K	train	7.05%	49.98%	8,380 M

Table 2: The applications used in our evaluation.

due to prefetching in a system without memory encryption. The second pair of bars shows the relative reduction in execution time due to predecryption in a system with memory encryption. If a prefetcher is only able to hide memory latency alone, the relative execution time in *Enc+CP+Sbuff* would be higher than that in *NoEnc+CP+Sbuff*. If a predecryptor is equally good at hiding decryption latency as it is at hiding memory latency, the relative execution time in *Enc+CP+Sbuff* would be similar to that in *NoEnc+CP+Sbuff*. Finally, if the prefetcher is better at hiding decryption latency than it is at hiding memory latency alone, the relative execution time *Enc+CP+Sbuff* would be lower than that in *NoEnc+CP+Sbuff*.

Our results in Figure 3 indicate that in most applications the prefetcher is at least as good at hiding decryption latency as it is at hiding memory latency, and in some applications it is noticeably better. This indicates that the prefetcher is indeed hiding some of the decryption latency in addition to the memory latency. In fact, the fraction of decryption latency that a prefetcher hides is often larger than the fraction of the memory latency it hides. Overall, without encryption, a prefetcher reduces execution time by 12% on average, while the reduction is 16% in a system with memory encryption. This indicates that predecryption is a promising scheme for alleviating the performance impact of memory encrytion.

However, one disappointment in our current setup is the lack of significant benefit from adding a correlation predictor to the stream buffers mechanism. To investigate this, we evaluate the performance of predictors in different configurations, and show the results in Figure 4.

Each bar has three segments. The bottom segment shows the number of L2 cache misses that are correctly predecrypted with the given predecryption setup. The full or partial miss latency (which includes memory latency and decryption latency) of these accesses is hidden and the data is read from the predecryption buffer. The middle segment shows the percentage of L2 cache misses that go to the main memory and also suffer the full decryption latency. The top segment shows the number of blocks that are predecrypted needlessly. All bars are shown relative

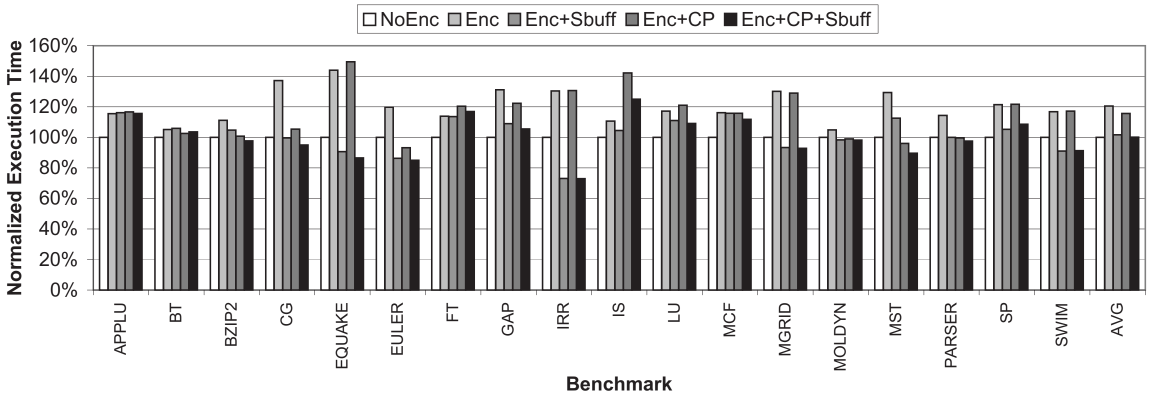

Figure 2: Application performance with different predecryption schemes

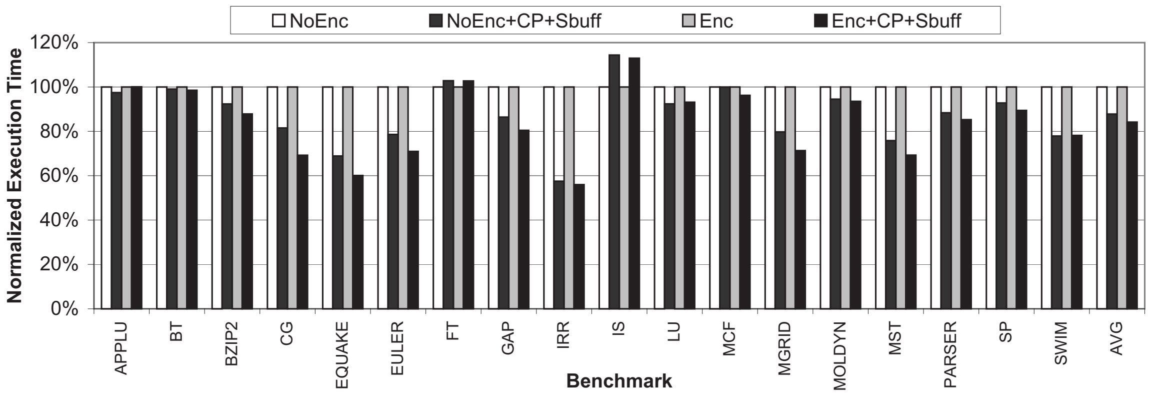

Figure 3: Predecryption benefit over standard prefetching

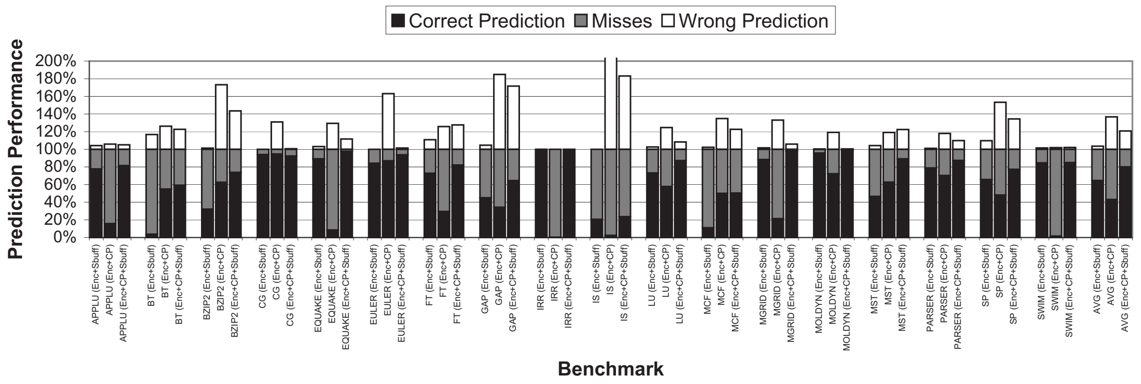

Figure 4: Prediction performance for different predecryption schemes

PARAMETERS

Processor	4 GHz, 6-way out-of-order issue Int, fp, ld/st FUs: 4, 3, 4 Branch penalty: 17 cycles ROB size: 248
Memory	L1-Inst: 16 KB, 2 way, 32-B line, WB, RT: 2 cycles, LRU, Outstanding ld+st misses: 16+16. L1-Data: 16 KB, 2 way, 64-B line, WB, RT: 3 cycles, LRU, Outstanding ld+st misses: 24+24. L2-Unif. (shared/per proc): 1 MB, 8-way, 64-B line, RT: 16 cycles, LRU, Outstanding ld+st: 24+24. Predecryption Buffer: 4-way, LRU, 64-B line, 16KB for Sbuff and 32KB for CP (48KB in Sbuff+CP) Memory bus: 1 GHz, 4-Byte wide, split-transaction RT memory latency: 275 cycles
Stream Buffers	Maximum 16 streams, 8 entries/buffer
Correlation Table	Replicated organization [18], 64K entries, 2-way, 2 levels, 2 successors/level, 8-cycle access, 647KB total
Encr/Decryption	128 cycles for each cache line

CONFIGURATIONS

NoEnc	No encryption/decryption delay and no predecryption
Enc	Encryption/decryption delay and no predecryption
Sbuff	No encryption/decryption delay and prefetching with stream buffers
Enc+Sbuff	Encryption/decryption delay and predecryption with stream buffers
CP	No encryption/decryption delay and prefetching with correlation table
Enc+CP	Encryption/decryption delay and predecryption with correlation table
CP+Sbuff	No encryption/decryption delay and prefetching with both stream buffers and correlation table
Enc+CP+Sbuff	Encryption/decryption delay and predecryption with both stream buffers and correlation table (SPEC2K benchmarks use delta scheme for the correlation table with this configuration)

Table 3: Parameters and configurations of the simulated architecture. Latencies correspond to contention-free conditions. *RT* stands for round-trip *from the processor*. We assume an AES encryption/decryption algorithm is used and can be performed in 128 cycles. This is roughly in line with 14-cycle latency on a 154 MHz processor reported in [16].

to the number of L2 misses in the baseline configuration. On average, the correlation predictor alone successfully predecrypts about 43% of the original L2 cache misses. However, this scheme also causes an extra 37% more predecryptions of data that is not useful to the application. These extra predecryptions could be the reason for the lack of performance benefit when the correlation predecryptor is added to the stream buffers. These extra memory accesses cause contention delays and interfere with memory requests for valid data. This is clearly evident in the *is* benchmark, where the addition of correlation prefetching slows down the execution of the benchmark, due to a high number of wrong predictions and low number of correct ones.

Future work will involve implementing a confidence scheme to only send predecryption requests for which there is some confidence that the data will be used. Another reason for low benefit of correlation prefetching is that the table may be too small for the data set size in some applications. We use a tagged correlation table and the lack of attempted predecryptions in some applications (e.g. *irr*) indicates that many misses fail to match an entry in the correlation table. However, a larger table may not be feasible to put on-chip, and our future work will focus on improving the table's hit rate while maintaining or further reducing its size. Our results show that stream buffers are very accurate in terms of predecrypting only useful data, and they also act as a filter that helps utilize the correlation table better. The combination of the two prediction mechanisms can predecrypt a very high percentage of L2 cache misses (80%), while only generating 20% additional unnecessary predecryptions.

Another interesting observation that can be made from Figures 2 and 4 is that in four benchmarks (applu, bt, ft, and mcf), we correctly predecrypt a large portion of L2 cache misses, but the execution compared with *Enc* is still similar. This is because many of the correct predecryptions in these applications only hide partial miss latency, and a significant fraction of the latency remains. This, combined with the increase in bus traffic due to mispredictions (top segment of each bar in Figure 4), results in a low overall benefit from predecryption. This confirms that reducing these extra predecryptions should make our scheme even more attractive.

5. Conclusions

Our results show that predecryption compares favorably to schemes such as one-time-pad because the execution time of some benchmarks can actually be reduced even with the extra latency. In schemes such as one-time-pad there will always be some slowdown in execution time because the decryption delay of a memory request can never be fully hidden as it can in our scheme. Since the data shows that the correlation predecryptor does not provide much performance improvement over simply stream buffers, we feel that this scheme will appear even more attractive once the correlation predecryptor implementation is optimized.

References

[1] T. Alexander and G. Kedem. Distributed Predictive Cache Design for High Performance Memory Systems. In *the 2nd Intl. Symp. on High-Performance Computer Architecture*, pages 254–263, 1996.

[2] M. J. Charney and A. P. Reeves. Generalized Correlation Based Hardware Prefetching. *Tech. Rep. EE-CEG-95-1, Cornell University*, 1995.

[3] T. F. Chen and J. L.Baer. Reducing Memory Latency via Non-Blocking and Prefetching Cache. In *the 5th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 51–61, 1992.

[4] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic. Memory-system design considerations for dynamically-scheduled processors. In *Proc. of the 24th Intl. Symp. on Computer Architecture*, 1997.

[5] G. Hinton and D. Sager and M. Upton and D. Boggs and D. Carmean and A. Kyker and P. Roussel. The Microarchitecture of the Pentium 4 Processor. *Intel Technology Journal*, (First Quarter), 2001.

[6] T. Gilmont, J.-D. Legat, and J.-J. Quisquater. Enhancing the Security in the Memory Management Unit. In *Proc. of the 25th EuroMicro Conf.*, 1999.

[7] A. Huang. *Hacking the Xbox: An Introduction to Reverse Engineering*. No Starch Press, San Francisco, CA, 2003.

[8] IBM. *IBM Power4 System Architecture White Paper*, 2002. http://www-1.ibm.com/servers/ eserver/pseries/hardware/whitepapers/power4.html.

[9] International Planning and Research Corporation. *6th BSA Global Software Piracy Study*, 2001. http://www. bsa.org/resources/2001-05-21.55.pdf.

[10] D. Joseph and D. Grunwald. Prefetching Using Markov Predictors. In *the 24th Intl. Symp. on Computer Architecture*, pages 252–263, 1997.

[11] N. Jouppi. Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers. In *the 17th Intl. Symp. on Computer Architecture*, pages 364–373, 1990.

[12] A. Lai, C. Fide, and B. Falsafi. Dead-Block Prediction and Dead-Block Correlating Prefetchers. In *the 28th Intl. Symp. on Computer Architecture*, pages 144–154, 2001.

[13] D. Lie, J. Mitchell, C. Thekkath, and M. Horowitz. Specifying and Verifying Hardware for Tamper-Resistant Software. In *IEEE Symp. on Security and Privacy*, 2003.

[14] D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz. Architectural Support for Copy and Tamper Resistant Software. In *Proc. of the 9th Intl. Conf. on Architectural Support for Programming Languages and Operating Systems*, 2000.

[15] S. Palacharla and R. Kessler. Evaluating Stream Buffers as a Secondary Cache Replacement. In *the 21st Intl. Symp. on Computer Architecture*, pages 24–33, 1994.

[16] P. Schaumount, H.Kuo, and I. Verbauwhede. Unlocking the design secrets of a 2.29 gb/s rijndel processor. In *Design Automation Conf.*, 2002.

[17] T. Sherwood, S. Sair, and B. Calder. Predictor-Directed Stream Buffers. In *the 33rd Intl. Symp. on Microarchitecture*, pages 42–53, 2000.

[18] Y. Solihin, J. Lee, and J. Torrellas. Using a user-level memory thread for correlation prefetching. In *29th Intl. Symp. on Computer Architecture (ISCA)*, 2002.

[19] Standard Performance Evaluation Corporation. Spec benchmarks. *http://www.spec.org*, 2000.

[20] G. Suh, D. Clarke, B. Gassend, M. van Dijk, and S. Devadas. Efficient Memory Integrity Verification and Encryption for Secure Processor. In *Proc. of the 36th Intl. Symp. on Microarchitecture*, 2003.

[21] J. Yang, Y. Zhang, and L. Gao. Fast Secure Processor for Inhibiting Software Piracy and Tampering. In *Proc. of the 36th Intl. Symp. on Microarchitecture*, 2003.