Operating System Controlled Processor–Memory Bus Encryption

Xi Chen, Robert P. Dick, and Alok Choudhary
Electrical Engineering and Computer Science Department
Northwestern University
2145 Sheridan Road
Evanston, Illinois 60208

***Abstract*—Unencrypted data appearing on the processor–memory bus can result in security violations, e.g., allowing attackers to gather keys to financial accounts and personal data. Although on-chip bus encryption hardware can solve this problem, it requires hardware redesign or increases processor cost. Application redesign to prevent sensitive data from appearing on the processor–memory bus is extremely difficult. We propose and evaluate a processor–memory bus encryption technique for embedded systems that requires no changes to applications or hardware. This technique exploits cache locking or scratchpad memory, features present in many embedded processors, permitting the operating system (OS) virtual memory infrastructure to automatically encrypt data belonging to protected processes as they are written to off-chip memory. Pages belonging to unprotected processes are stored unencrypted to prevent performance and energy consumption penalties.**

We evaluate the proposed bus encryption technique using full system simulation. Experimental results indicate that it is possible to prevent the working data sets of processes from appearing on the processor–memory bus in plaintext, without using dedicated hardware and without changing applications. The OS based technique results in 1.37× slowdown for protected processes for processors with 512 KB of L2 cache and 1.78× slowdown for processors with 256 KB of L2 cache. There are negligible performance penalties for unprotected processes.

I. Introduction

Most embedded systems exchange data between processor and external memory in plaintext. These data may be confidential, e.g., keys protecting account information or commercial transaction information. An adversary can access them by observing bus signals using a logic analyzer or inexpensive FPGA-based monitoring hardware [1, 2]. We refer to this as a *bus snooping* attack. Although some embedded system manufacturers add security features to their products, few protect the processor–memory bus; it is one of the most vulnerable points in an embedded system.

As a result of the increasing use of embedded systems for high-security applications such as electronic commerce, the importance of protecting the processor–memory bus is increasing. Achieving this by rewriting existing applications is difficult; explicit control of all processor–memory transactions of protected processors would be necessary. To address this problem, Best [3] proposed a technique called *bus encryption*, in which the processor encrypts data before transfer to external memory and decrypts data after fetching them. This technique has previously required the use of application-specific encryption hardware that is either slow or imposes high area overhead, limiting commercial use. We propose a novel software-based technique that encrypts data on the processor–memory bus without the addition of bus encryption hardware and without changes to applications.

This work was supported in part by AFOSR award FA9550-06-1-0152 and in part by NSF awards ITR-CCR-0325207 and CNS-0347941.

II. Past Work and Contributions

Since the idea of bus encryption appeared 27 years ago, there have been numerous academic evaluations and industrial implementations [4]. However, all have been implemented with and rely on additional special-purpose hardware.

A number of hardware-based bus encryption patents exist. Takahashi et al. [5] proposed embedding a secure direct memory access (DMA) controller in the same chip with a microprocessor core, an internal memory, and an encryption/decryption logic. The secure DMA controller manages all CPU external requests, while data transmission between external and internal memory uses the encryption/decryption engine. Candelore et al. [6] proposed a hardware-based method to transfer program information encrypted using cipher block chaining between external memory and a secure circuit, e.g., a cryptographic integrated circuit with bus encryption hardware. The DS5240 and DS5250 [7] produced by Dallas Semiconductor integrate block cipher engines to encrypt data on the processor–memory bus using DES or triple-DES encryption. Bus encryption hardware is used in applications such as pay-television control and credit cards. In-package encryption/decryption coprocessors [8] have also been proposed for bus encryption in embedded systems.

Researchers have proposed architectural advances to support bus encryption. Guilmont et al. [9] proposed an improved memory management unit, called an SMU, as a hardware solution to several security problems. The SMU includes a pipelined block cipher unit. DES and 3-DES are used for encryption. The XOM project [10] proposes the hardware implementation of execute-only memory that is divided into different compartments to protect secure processes from insecure ones. A symmetric cipher, such as AES, is used for hardware bus encryption. AEGIS is an architecture for building computing systems that are secure against physical and software attacks [11]. The cryptographic engine achieves reasonable performance at a relatively high hardware cost (see Section V for additional details).

In summary, hardware-based approaches may counter bus snooping attacks in embedded systems. However, they require additional hardware components and impose significant overhead. Therefore, their use increases the design time and cost of embedded systems.

The proposed OS controlled software-based processor–memory encryption technique has the following characteristics:

1) Unlike existing techniques, it requires no hardware modification and therefore no system redesign;
2) It requires no changes to applications;
3) It maintains moderate performance for protected processes and unchanged performance for normal processes; and
4) With the exception of a few low-level cache control rou-

978-3-9810801-3-1/DATE08 © 2008 EDAA

tines, much of the design is portable among architectures that support cache locking or using cache as scratchpad memory, i.e., most embedded processors as well as other processors frequently used in real-time applications.

The proposed bus encryption technique can be used to enhance security in many applications. For example, it can prevent an attacker from gathering information revealed on the bus (e.g., biometric authentication data); it can prevent a malicious user from using bus snooping to obtain data for other users of a multi-user system; and it can protect bank customers from dishonest bank employees attempting to obtaining PINs via bus snooping. The proposed bus encryption technique prevents these, and other attacks relying on reading unprotected data appearing on the processor–memory bus, since data transmitted on the processor–memory bus are encrypted.

III. Operating System Controlled Processor–Memory Bus Encryption

In this section, we give an overview of the proposed software bus encryption technique and explain some notable design requirements in more detail. The proposed technique is designed for systems with on-chip caches, memory management units (MMUs), and external memory. We make the following assumptions:

1) *Environment*: The processor is trusted and everything outside the processor is prone to monitoring. Depackaging and reverse engineering the processor are difficult; we assume that attackers do not have access to cache data.

2) *Cache*: The cache supports *locked mode*, i.e., lines may be explicitly locked into the cache or it may be used as scratchpad memory, i.e., a portion of the cache may be used as addressable memory. In either case, the OS controls the transfer of data between cache and memory. Many off-the-shelf processors used in consumer electronics, e.g., processors in the Intel XScale family [12], support these features.

3) *Initialization Vector*: A 32-bit vector is randomly generated during each page encryption. We use "/dev/urandom" as our random pool. The vector is then padded with zeros to 128 bits and used as the initialization vector for AES, which is described in Section III-B. After each page is encrypted, the vector is stored along with the encrypted page in the off-chip memory. This approach is similar to that described by Suh et al. [11]. The memory overhead for the random vector is 0.4% in the worst case, i.e., when the page size is 1 KB.

4) *Key*: A single encryption key is generated each time the system resets. This poses no special problems: no encrypted data pages need to be maintained from boot to boot. The key is stored on-chip and is only known to the OS. Note that every secure page in off-chip RAM is encrypted. Since the attacker is unable to learn either the plaintext or the secret key, every secure page can be safely encrypted using a randomly-generated initialization vector described in (3) and a key that is randomly generated on system startup. Although many hardware-based bus encryption techniques assume the secret key is stored in on-chip non-volatile memory, this is not a requirement in our technique.

5) *Attacker*: We assume attackers belong to Class I (clever outsiders) in the IBM taxonomy [13]. They have physical access to the device and are able to monitor the bus. However, they are unable to tamper with the contents of memory, or modify the kernel, which will often reside in read-only nonvolatile memory soldered to the printed circuit board.

Figure 1 gives an overview of a system in which the proposed bus encryption technique is used. The cache dynamically switches between locked and unlocked mode depending on whether the currently-running process is protected. The data memory pages of protected applications are stored in off-chip RAM only in encrypted form. Every time a *secure page*, i.e., a page belonging to a protected process, is brought into off-chip RAM from the swap device, it is decrypted by the OS and mapped into the cache, i.e., the data for protected processes are stored in plaintext only when in cache. Pages belonging to unprotected processes are stored unencrypted to minimize performance degradation. Note that our technique requires no additional hardware. It does require MMU and cache locking or cache-as-memory functionality. Some embedded processors only allow part of the cache to be used in locked mode. For such processors, only the portion of cache supporting explicit management can be considered secure.

Figure 2 illustrates the operation of the proposed OS-based processor–memory bus encryption technique. The CPU interacts with the cache, which in turn communicates with external memory through the memory controller. Data transmitted on the processor–memory bus are encrypted if they belong to protected processes. When the CPU generates a page access request, virtual address translation, cache line replacement, OS-controlled encryption/decryption, memory read/write, and OS-controlled swapping may be required. Among these, cache line replacement due to conflict misses contributes most to performance penalties.

To explain the proposed technique more clearly, we trace the events that occur during a page access request (see Figure 2). When the current process issues a page access request, the virtual address generated by the CPU first uses the MMU for address translation. If the virtual address is invalid, the MMU generates an exception and traps to the OS error handler. Otherwise, the legal virtual address is translated to a physical address that is used to access data in the cache. Unprotected processes are thus prevented from accessing secure pages. Page contents are fetched from cache to CPU immediately on a cache hit; on a cache miss, the required page must be read from external RAM.

When the target page is in off-chip memory, a page fault occurs when the active process is a secure process. The OS checks whether it is a secure page and, if so, decrypts it. The OS page handler then moves the referenced page into the cache from off-chip memory. The cache operates normally when unprotected processes are active, i.e., no encryption, decryption, or cache locking is used. When a secure page is evicted from the cache, the OS encrypts the page before sending it to off-chip memory. The general idea of mapping swapped data back to a special region of memory instead of backing store is somewhat counterintuitive but has been validated via a complete implementation [14].

As we can see from Figure 2, unprotected processes are unable to access secure pages. The data of secure processes are encrypted before transfer to off-chip memory thereby preventing plaintext information from appearing on the processor–memory bus. Unprotected processes are able to execute normally without additional overhead.

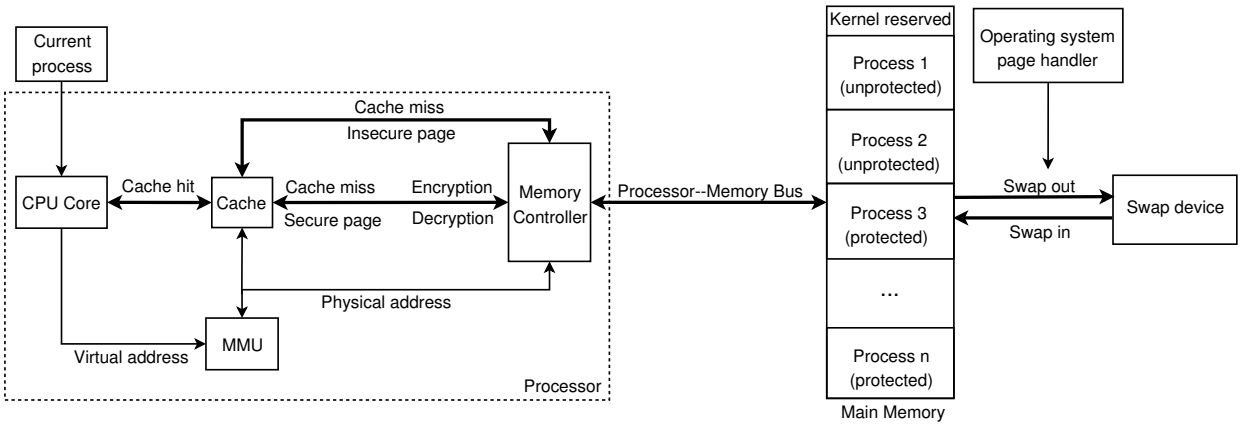

Figure 1. System architecture.

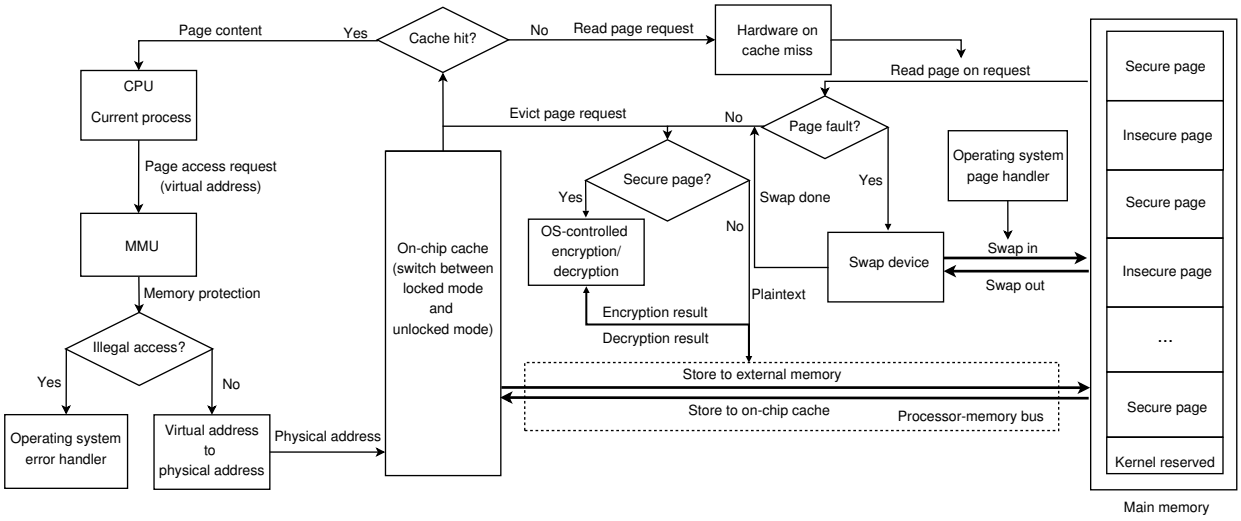

Figure 2. Operation of the OS-controlled processor–memory bus encryption technique.

III.A. Operating System Changes

The major changes to the OS kernel involve page handling and control of cache status.

Modifications to the virtual memory infrastructure will result in special handling for page mapping and page faults in protected applications. The OS is responsible for correct page mapping between virtual memory and physical memory. An additional Boolean variable is added to each process descriptor, which resides in kernel address space, to indicate whether the corresponding process is protected. Once a process is created, the OS sets the variable depending on application initialization information securely stored in the system. Unprotected processes can access only unprotected pages, to prevent plaintext information leakage. Protected processes may access both secure and unprotected pages within their address spaces.

Special actions are required during page faults, as shown in Figure 3. The proposed technique supports a swap device, but does not require one. When a secure page is swapped in, it will always be mapped to a physical page in the cache and decrypted by the kernel. Similarly, when a page belonging to a secure process is evicted, either as a result of the requirement to free space in secure memory such as on-chip cache or as the result of a context switch from a protected process, the secure page is encrypted and swapped out. Therefore, the page handler in the proposed technique uses kernel cryptographic routines to encrypt and decrypt pages.

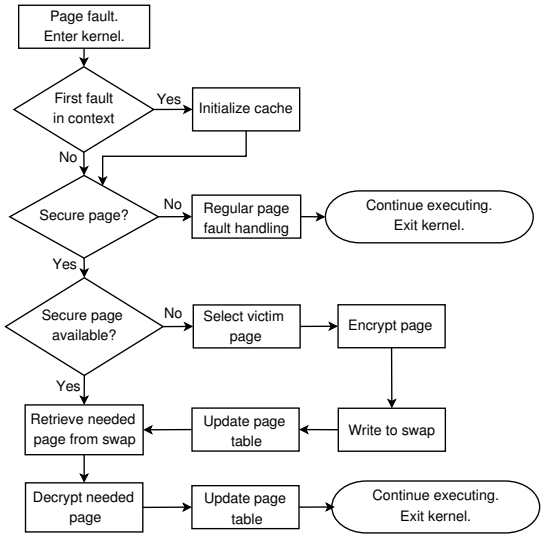

Figure 3. Page fault handling flowchart.

When a context switch to a protected process occurs, the

OS places the cache in locked mode, i.e., a region of the cache is locked to a region of external memory and placed in write-back mode to prevent protected data from leaking. As a result, the locked region of the cache is used as secure memory; the OS determines when the contents stored in this region of cache are written back to RAM. Therefore, we can prevent sensitive data in protected processes from ever appearing on the bus in plaintext. The locked region holds encryption/decryption code variables and several pages of data for secure applications.

On context switches from protected processes to other processes, the OS invalidates or zero-fills the cache to prevent data in the protected region from being written back to external memory. Recall that the main memory address range associated with the locked region of cache is not used for normal data storage. The cache operates normally for unprotected processes, enabling them to execute with full performance.

III.B. Encryption Algorithm

We considered a number of symmetric and asymmetric key encryption algorithms [15] for use in the proposed technique. We selected AES in cipher block chaining (CBC) mode because it is generally believed to achieve high security and has adequate performance for this application. Note that one can trade off security for performance, e.g., by using AES in counter mode. We evaluated the performance of AES in CBC mode on a 1 GHz AMD processor using OpenSSL 0.9.8d [16]. Experimental results indicate that encrypting a 1,024 B block and a 4,096 B block require 10,620 and 42,118 cycles, respectively. Decrypting an input block requires approximately the same number of cycles as encrypting the block, since the same routine is used for encryption and decryption.

III.C. Importance of Page Size

The MMU plays a critical role in separating protected processes from unprotected processes in virtual-to-physical address translation. Its settings also determine the page size of the system. The encrypted data of protected processes are exchanged between off-chip memory and secure cache in page-sized units. In other words, the line size in the software-managed cache is equal to the page size. Therefore, the size of a page has great impact on the performance overhead of the proposed technique because it determines the line size and total number of cache lines when the cache is in locked mode. The page size is determined by the OS and MMU. The Intel XScale processor family implements the ARM Version 5TE instruction set architecture. Therefore, XScale processors support page sizes ranging from 1 KB to 1 MB [12]. Linux has been ported to the ARM platform [17]. This port supports 2 KB to 32 KB page sizes, although the kernel sources might need patching to support smaller page sizes such as 1 KB. Although small page sizes, e.g., 1 KB, may increase the overhead associated with page management, this overhead is low in most embedded systems, which commonly have limited physical memory ranging from 16 MB to 64 MB, with a corresponding memory overhead ranging from 64 KB to 256 KB given a two-level paging scheme. This 0.4% overhead is negligible compared to physical RAM size. In addition, the first-level and second-level page tables need not be present in memory at the same time, thus further reducing the memory management overhead. Many embedded processors, e.g,. PowerPC 400 family processors [18], support a small page size such as 1 KB.

IV. Evaluation

In this section, we present performance data for applications running on a simulated 800 MHz Pentium 4 processor with and without protection using the proposed bus encryption technique. The proposed technique degrades the performance of protected threads due to an increase in conflict misses resulting from the larger effective cache line size due to software cache management. The proposed technique is appropriate in cases where embedded systems designers can tolerate a significant performance overhead for protected processes. This overhead is offset by other special advantages, i.e., prevention of bus snooping for data sets used by protected processes, requiring no changes to hardware, and requiring no changes to applications. In this section we indicate the performance penalties of using the proposed technique under different circumstances, e.g., different page sizes and different cache sizes. Our simulation results indicate that it is possible to protect the data accessed by a process from appearing on the processor–memory bus without dedicated hardware and without changes to applications. For processors supporting 1 KB page size, the proposed technique results in 1.37× slowdown for protected processes given 512 KB of L2 cache and 1.78× slowdown given 256 KB of L2 cache.

IV.A. Simulation and Evaluation Environment

We use Simics [19] for system-level instruction set simulation. Our goal is to determine whether the relative overhead of the proposed technique is low enough to permit practical use. The simulated system has separate instruction and data caches at level 1 backed by a unified level 2 cache. Although the proposed technique will be used with embedded processors, e.g., ARM processors, we connect this cache hierarchy to an 800 MHz Pentium 4 processor because it is better supported by Simics. The relative overhead is likely to be similar for ARM and x86 processors because they have similar memory hierarchies and relative memory access timing characteristics.

Characteristics of level 1 and level 2 cache are listed in Table I. In addition, dynamic RAM (DRAM) usually has an access time ranging from 5 ns to 70 ns. Therefore, we set the main memory access time to 100 processor cycles.

Our analysis shows that the performance impact of context switches on unprotected processes. We multiply the number of L2 cache lines by the time to refill each line to calculate the time to flush the L2 cache on context switches from protected processes. The flush time is approximately 0.5 ms for a 256 KB L2 cache, which is small compared to the average timeslice of 210 ms for a running task assigned by the Linux 2.4 scheduler [20]. Therefore, cache effects triggered by context switches from protected processes to unprotected processes have little impact on the performance of unprotected processes.

IV.B. Simulation Process

To model the cache hierarchy of commonly-available ARM-based embedded systems using the proposed bus encryption technique, we have changed the L2 cache as follows:

1) During the execution of a secure process, the L2 cache is used as scratchpad memory to minimize the performance impact of encrypting/decrypting secure pages. Although normally only part of the cache, e.g., 7/8, may be used as scratchpad memory in Intel XScale processors [21], Simics supports only

TABLE I
L1 & L2 CACHE CHARACTERISTICS

Name	L1 Instruction Cache	L1 Data Cache	L2 Cache
Default Cache Size (KB)	16	8	512
Line Number	256	128	8192
Line Size (B)	64	64	64
Writing Method	Write-through	Write-through	Write-back
Mapping Mechanism	8-way	4-way	8-way
Replacement Mechanism	LRU	LRU	LRU
Cycles on read-hit	1	2	7
Cycles on write-hit	1	2	7

cache sizes that are powers of two. Therefore, we use the whole cache as scratchpad memory for the convenience of simulation. This has little impact on the results because access time reduction permitted by the extra 1/8 cache as scratchpad memory is similar to that by using it as normal cache.

2) To determine the relationship between performance and cache configuration, we considered cache sizes of 64 KB, 128 KB, 256 KB, and 512 KB.

3) The read miss and write miss penalties were increased in order to compensate for the increased line size. These penalties were then added to the number of cycles required to generate a 32-bit random vector and encrypt or decrypt a page in order to determine the cost of encryption or decryption.

4) The cache line size is set to the page size, i.e., 1 KB and 4 KB. Therefore, a cache line replacement results in a page-sized data transmission on the processor–memory bus. Note that a cache miss suffers from the performance penalty for encrypting/decrypting a page as indicated above (3). This implies that secure pages are encrypted before being transferred to external memory and decrypted when transferred into the secure cache region.

5) The cache is set to fully-associative because its contents are managed by the OS.

We use programs from Mediabench [22] to determine the performance impact of the proposed technique on a number of simulated memory organizations. We choose AES as our encryption algorithm for the reasons indicated in Section III-B. We evaluated all the benchmarks that were compatible with the libraries on the default Simics filesystem image, which runs the 2.4.18 Linux kernel. Table II shows the experimental results for 1 KB and 4 KB pages. Each row shows the results for an application in Mediabench given different cache sizes and page sizes. The *normal* column specifies the execution time when the application is unprotected, while the *protected* column shows the execution time when the application is protected. We used stalling simulation mode because it is fast and maintains reasonable accuracy.

IV.C. Analysis

This section explains the impact of page size and cache size on the performance of the proposed technique.

IV.C.1) Influence of Page Size: Figure 4 illustrates the influence of page size on the performance overhead of the proposed technique. We compare the average performance overhead using 1 KB and 4 KB pages. When 1 KB pages are used, our technique results in a 1.37× slowdown of protected processes for a 512 KB L2 cache and a 1.78× slowdown for a 256 KB L2 cache. When 4 KB pages are used, our technique results in 1.74× slowdown of protected processes also for a 512 KB L2 cache and a 2.53× slowdown for a 256 KB L2 cache. For 64 KB L2 caches, the performance penalty is 2.44× and 8.50× slowdown for 1 KB and 4 KB page sizes, respectively. The results indicate that the performance overhead is lower with a smaller page size. This can be explained as follows. Reducing the page size increases the number of pages, thereby reducing conflicts due to accessing portions of pages.

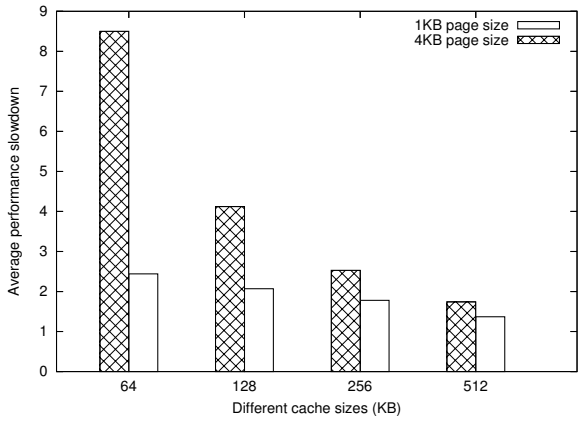

Figure 4. Influence of page and cache sizes on performance overhead.

IV.C.2) Influence of Cache Size: Figure 4 also shows the influence of cache size on the performance overhead of the proposed technique. We choose L2 caches with different sizes, e.g., 64 KB, 128 KB, 256 KB, and 512 KB for evaluation. As we can see from the figure, when the L2 cache is relatively small, e.g., 64 KB, the simulated system suffers a high performance degradation with both 1 KB and 4 KB page sizes. As cache size increases, the performance overhead of the proposed technique decreases. Specifically, for 1 KB page size, when the cache size grows from 64 KB to 128 KB, the penalty decreases from an average of 2.44× to an average of 2.07×. Therefore, we conclude that the average slowdown decreases as the L2 cache size increases due to more cache hits and thus less time spent on decryption, encryption, and accessing memory. The overhead of the proposed technique is lowest for smaller page sizes and larger caches.

V. COMPARISON WITH HARDWARE IMPLEMENTATION OF ENCRYPTION ALGORITHMS

Designing bus encryption hardware for symmetric encryption/decryption algorithms such as AES in CBC mode is challenging. High data rates are required for the processor–memory bus. CBC is generally used for high-security applications. However, pipelining it to permit high data rates is very difficult. To the best of our knowledge, there are no commercial hardware encryption engines in embedded systems that implement AES in cipher block chaining mode at a high enough data rate for use in processor–memory bus encryption. The DS5240 and DS5250 produced by Dallas Semiconductor [7] only implement DES and triple-DES with a low maximum frequency, i.e., 25 MHz. The Intel IXP422 and IXP425 processors have additional network processor engines that support AES in CBC mode. However, the encryption speed of 70 Mb/s is far less than the processor–memory bus data rate of 4,256 Mb/s, given a 32-bit, 133 MHz bus. This makes the network encryption engines inappropriate for bus encryption [23, 24].

TABLE II
EXPERIMENTAL RESULTS FOR SYSTEMS WITH 1 KB AND 4 KB PAGES

cache size	64 KB					128 KB					256 KB					512 KB				
penalty	normal (s)	protected (s)		slowdown (×)		normal (s)	protected (s)		slowdown (×)		normal (s)	protected (s)		slowdown (×)		normal (s)	protected (s)		slowdown (×)	
		1 KB page	4 KB page	1 KB page	4 KB page		1 KB page	4 KB page	1 KB page	4 KB page		1 KB page	4 KB page	1 KB page	4 KB page		1 KB page	4 KB page	1 KB page	4 KB page
adpcmenc	0.03	0.11	0.67	3.67	22.33	0.03	0.08	0.21	2.67	7.00	0.03	0.07	0.12	2.33	4.00	0.03	0.06	0.09	2.00	3.00
adpcmdec	0.03	0.11	0.70	3.67	23.33	0.03	0.08	0.21	2.67	7.00	0.03	0.07	0.12	2.33	4.00	0.03	0.05	0.08	1.67	2.67
epic	0.17	0.56	0.96	3.29	5.65	0.16	0.49	0.64	3.06	4.00	0.16	0.43	0.49	2.69	3.06	0.16	0.22	0.28	1.38	1.75
unepic	0.05	0.24	0.49	4.80	9.80	0.04	0.20	0.33	5.00	8.25	0.04	0.17	0.22	4.25	5.50	0.04	0.08	0.11	2.00	2.75
g721enc	0.73	0.83	1.35	1.14	1.85	0.73	0.78	1.06	1.07	1.45	0.73	0.78	0.80	1.07	1.10	0.73	0.75	0.78	1.03	1.07
g721dec	0.69	0.79	1.31	1.14	1.90	0.69	0.75	1.03	1.09	1.49	0.69	0.73	0.81	1.06	1.17	0.69	0.72	0.74	1.04	1.07
gs	3.65	6.11	17.62	1.67	4.83	3.63	4.90	10.85	1.35	2.99	3.63	4.49	5.67	1.24	1.56	3.63	4.20	4.42	1.16	1.22
gsmenc	0.39	0.50	1.06	1.28	2.72	0.39	0.44	0.68	1.13	1.74	0.39	0.43	0.47	1.10	1.21	0.39	0.42	0.44	1.08	1.13
gsmdec	0.19	0.27	0.71	1.42	3.74	0.19	0.24	0.43	1.26	2.26	0.19	0.23	0.28	1.21	1.47	0.19	0.22	0.24	1.16	1.26
jpegenc	0.07	0.18	0.49	2.57	7.00	0.07	0.14	0.29	2.00	4.14	0.07	0.11	0.17	1.57	2.43	0.07	0.09	0.12	1.29	1.71
jpegdec	0.03	0.10	0.39	3.33	13.00	0.03	0.08	0.21	2.67	7.00	0.03	0.06	0.11	2.00	3.67	0.03	0.05	0.08	1.67	2.67
mipmap	0.46	0.80	2.98	1.74	6.48	0.46	0.69	1.28	1.50	2.78	0.46	0.66	0.79	1.43	1.72	0.46	0.63	0.67	1.37	1.46
osdemo	0.24	0.82	2.38	3.42	9.92	0.24	0.64	1.46	2.67	6.08	0.24	0.51	0.85	2.13	3.54	0.24	0.36	0.50	1.50	2.08
texgen	0.60	1.86	9.66	3.10	16.1	0.60	1.49	3.68	2.48	6.13	0.60	1.01	2.05	1.68	3.42	0.60	0.82	1.02	1.37	1.70
mpeg2enc	3.12	3.91	7.06	1.25	2.26	3.10	3.60	4.71	1.16	1.52	3.10	3.50	3.77	1.13	1.22	3.10	3.35	3.45	1.08	1.11
mpeg2dec	0.48	0.73	2.41	1.52	5.02	0.48	0.62	1.02	1.29	2.13	0.48	0.58	0.69	1.21	1.44	0.48	0.54	0.60	1.13	1.25
average				2.44	8.50				2.07	4.12				1.78	2.53				1.37	1.74

Some researchers have proposed hardware implementations of AES in cipher block chaining mode. In AEGIS [11], a fully parallel AES implementation encrypts/decrypts a 64 B cache line in 10 processor cycles with an area cost of 300,000 gates. The performance overhead induced by the cryptographic engine is estimated to be as low as 25%. Therefore, the hardware-based approach is 1.42× faster than our software-based method, given 1 KB page size and 256 KB cache size, at the expense of 300,000 additional gates, almost four times the size of an ARM7EJ-S processor core. In summary, there are many applications in which the reduced design complexity and low cost of software-based processor–memory bus encryption may make it an attractive alternative to a hardware-based solution. This is the first article to propose a software-based solution.

VI. CONCLUSIONS

In this paper, we have presented a software-based bus encryption technique for use in embedded systems. This technique requires no changes to hardware or applications. This is a substantial advantage in terms of design complexity and integrated circuit area overhead. Our technique has been evaluated via full system simulation with a set of applications from Mediabench [22]. Experimental results indicate that the performance overhead for protected processes is 1.78× in systems with 256 KB L2 cache and 1.37× in systems with 512 KB L2 cache. Unprotected processes experience negligible overhead. Therefore, the approach appears to be applicable on embedded systems in which unprotected processes account for the majority of computational demand and for embedded systems with large caches and/or small page sizes that run computationally-intensive protected processes.

REFERENCES

[1] L. Whetsel, "An IEEE 1149.1 based logic/signature analyzer in a chip," in *Proc. Int. Test Conf.*, Oct. 1991, pp. 869–878.

[2] "I2C bus monitor," http://www.jupiteri.com.

[3] R. M. Best, "Microprocessor for executing enciphered programs," US Patent No. 4,168,396, Sept. 1979.

[4] R. Elbaz, et al., "Hardware engines for bus encryption: a survey of existing techniques," in *Proc. Design, Automation & Test in Europe Conf.*, Mar. 2005, pp. 40–45.

[5] R. Takahashi and D. N. Heer, "Secure memory management unit for microprocessor," U.S. Patent (from VLSI Technology, Inc.) No. 5,825,878, Oct. 1998.

[6] B. Candelore and E. Spunk, "Secure processor with external memory using block chaining and block reordering," U.S. Patent (from General Instrument Corporation) No. 6,061,449, May 2000.

[7] DS5240 and DS5250. Dallas Semiconductor (Maxim). http://www.maxim-ic.com/Microcontrollers.cfm.

[8] The IBM PCI Cryptographic Coprocessor. IBM Corporation. http://www.ibm.com/security/cryptocards.

[9] T. Gilmont, J.-D. Legat, and J.-J. Quisquater, "Enhancing security in the memory management unit," in *Proc. Euromicro Conf.*, Sept. 1999, pp. 449–456.

[10] D. Lie, et al., "Architectural support for copy and tamper resistant software," in *Proc. Int. Conf. Architectural Support for Programming Languages and Operating Systems*, Nov. 2000.

[11] G. E. Suh, et al., "AEGIS: Architecture for tamper-evident and tamper-resistant processing," in *Proc. Int. Conf. Supercomputing*, June 2003, pp. 160–171.

[12] Intel XScale Core Developer's Manual. Intel Corporation. http://download.intel.com/design/intelxscale/27347302.pdf.

[13] D. G. Abraham, et al., "Transaction security system," *IBM Systems*, vol. 30, no. 2, pp. 206–229, 1991.

[14] L. Yang, et al., "On-Line Memory Compression for Embedded Systems," *ACM Trans. Embedded Computing Systems*, to appear.

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

[16] "OpenSSL Release 0.9.8d," http://www.openssl.org.

[17] "ARM Linux Project," http://www.arm.linux.org.uk.

[18] PowerPC Processors. IBM Corporation. http://www-03.ibm.com/chips/power/powerpc/newsletter/pdf/oct2000.pdf.

[19] "Simics," http://www.virtutech.com.

[20] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, 2nd ed. O'Reilly & Associates, Inc., Dec. 2002.

[21] Intel XScale Technology. Intel Corporation. http://www.intel.com/design/intelxscale.

[22] C. Lee, M. Potkonjak, and W. H. M. Smith, "Mediabench: A tool for evaluating and synthesizing multimedia and communications systems," http://cares.icsl.ucla.edu/MediaBench.

[23] Intel IXP42X Processors Datasheet. Intel Corporation. http://download.intel.com/design/network/datashts/25247906.pdf.

[24] Intel IXP42X Processors Manual. Intel Corporation. http://download.intel.com/design/network/manuals/25248005.pdf.