Memory Encryption: A Survey of Existing Techniques

MICHAEL HENSON and STEPHEN TAYLOR, Dartmouth College

Memory encryption has yet to be used at the core of operating system designs to provide confidentiality of code and data. As a result, numerous vulnerabilities exist at every level of the software stack. Three general approaches have evolved to rectify this problem. The most popular approach is based on complex *hardware enhancements*; this allows all encryption and decryption to be conducted within a well-defined trusted boundary. Unfortunately, these designs have not been integrated within commodity processors and have primarily been explored through simulation with very few prototypes. An alternative approach has been to augment existing hardware with *operating system enhancements* for manipulating keys, providing improved trust. This approach has provided insights into the use of encryption but has involved unacceptable overheads and has not been adopted in commercial operating systems. Finally, *specialized industrial devices* have evolved, potentially adding coprocessors, to increase security of particular operations in specific operating environments. However, this approach lacks generality and has introduced unexpected vulnerabilities of its own. Recently, memory encryption primitives have been integrated within commodity processors such as the Intel i7, AMD bulldozer, and multiple ARM variants. This opens the door for new operating system designs that provide confidentiality across the entire software stack outside the CPU. To date, little practical experimentation has been conducted, and the improvements in security and associated performance degradation has yet to be quantified. This article surveys the current memory encryption literature from the viewpoint of these central issues.

Categories and Subject Descriptors: B.3.m [**Hardware**]: Memory Structures—*Miscellaneous*; C.1.0 [**Processor Architectures**]: General; C.4 [**Computer Systems Organization**]: Performance of Systems—*Reliability, availability, and serviceability*; D.4.2 [**Operating Systems**]: Storage Management—*Main memory*

General Terms: Design, Experimentation, Performance, Security

Additional Key Words and Phrases: Secure processors, memory encryption, confidentiality, protection, hardware attacks, software attacks

ACM Reference Format:
Michael Henson and Stephen Taylor. 2014. Memory encryption: A survey of existing techniques. ACM Comput. Surv. 46, 4, Article 53 (March 2014), 26 pages.
DOI: http://dx.doi.org/10.1145/2566673

BACKGROUND AND MOTIVATION

Encryption has been an important part of secure computing for decades, first in the DoD and national agencies and then publicly beginning with DES and public-key encryption in 1977 [Mel and Baker 2001]. As public use of computers continued to grow, so did the need to secure sensitive information. In 1991, Phil Zimmerman released the first version of Pretty Good Privacy (PGP), allowing anyone to encrypt e-mail and files. In

This material is based on research sponsored by the Defense Advanced Research Projects Agency (DARPA) under agreement number FA8750-09-1-0213.
Authors' addressess: M. Henson and S. Taylor, Thayer School of Engineering at Dartmouth College; emails: Michael.Henson@Dartmouth.edu and Stephen.Taylor@Dartmouth.edu.

DOI: http://dx.doi.org/10.1145/2566673

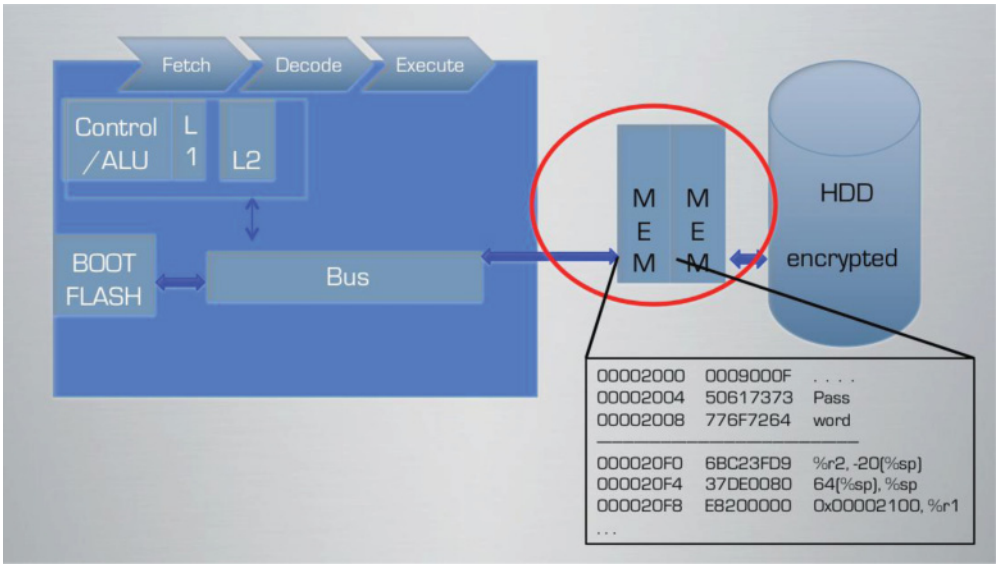

Fig. 1. System with full disk encryption but vulnerable code and data.

1995, Netscape developed the Secure Sockets Layer (SSL) protocol, combining public and private-key encryption to protect online financial transactions.

Full Disk Encryption (FDE) in commodity computer systems is a more recent innovation that provides confidentiality of all data stored on disk. Recent advances to the overall speed of processors, thanks to the march of Moore's law, and hardware-based encryption have resulted in several commercially viable FDE implementations. Software approaches to FDE include TrueCrypt, PGPDisk, FileVault, and Bitlocker. In addition, multiple hard drive manufacturers offer Self-Encrypting Drives (SEDs) in which encryption is handled entirely by the hard drive microcontroller. Several factors have resulted in increasing adoption of FDE technologies [Brink 2009]. Regulations, such as Sarbanes-Oxley and the Health Insurance Portability and Accountability Act (HIPAA), have increased the requirement for privacy. The advent of mobile computing and widespread movement of information over the Internet have raised concerns regarding physical access to data. Finally, numerous data breaches have been publicized, raising awareness of vulnerabilities.

Unfortunately, even with FDE, systems exhibit a major weakness in that data and code stored in memory are unencrypted (i.e., stored in the clear), as shown in Figure 1. This weakness has been exploited to gather encryption keys, passwords, passphrases, and other personal information from memory, thereby diminishing, or in some cases nullifying, the value of FDE [Halderman et al. 2008]. Since code is also stored in memory, it is possible to inject a wide variety of malicious implants into both user process and operating system kernels. Even applications designed specifically with security in mind have been shown to be vulnerable. For example, cryptographic libraries have been designed to prevent access to keys by zeroizing (or overwriting with zeros) a key after it has been used. This zeroizing of code is sometimes removed by compiler optimization because it appears superfluous, reintroducing the vulnerability [Chow et al. 2004].

To exploit memory vulnerabilities, numerous attack vectors have been developed. In a cold boot attack, for example, memory is frozen using a refrigerant and then removed from the computer. It is then quickly placed into a specially designed system

that reads out its content, targeting encryption keys and other sensitive information. Although this approach is novel, the idea of recovering encryption keys from memory has been described as early as 1998 [Kaplan 2007]. Even without cooling, some information persists in RAM for several minutes [Halderman et al. 2008]. However, cooling slows down the rate of data loss, reducing recovery errors [Chhabra and Solihin 2011]. Some approaches, such as the DMA-firewire attack, deliberately bypass FDE to enable forensic analysis. Unfortunately, these techniques are equally accessible to criminal organizations and other attackers, as well as legitimate law enforcement. Similar results are available via simple software attacks involving buffer overflows [Rabaiotti and Hargreaves 2010]. One particularly effective attack, bus-snooping and injecting, allows information to be captured or inserted via the bus lines between system components [Boileau 2006]. This exploitation method has been used to undermine the Xbox gaming system. This system was specifically designed to provide a secure chain of trust for enforcing Digital Rights Management (DRM). Bus-snooping was used to capture keys as they transited between read-only-memory and the CPU. These keys were then used to decrypt the secure boot loader undermining the entire chain of trust. Subsequently, low-cost "mod" chips were developed that can be soldered into the gaming system bus, allowing a user to bypass DRM restrictions and play pirated games [Steil 2005]. Alternatively, the same chips can be used to run various operating systems on the gaming system, allowing it to be used for illicit purposes [Rabaiotti and Hargreaves 2010].

Fortunately, access to information in conventional dynamic RAM presents an adversary with only a *fleeting* opportunity to obtain sensitive information between power cycles. However, dynamic RAM is being augmented or replaced with new nonvolatile alternatives—flash memory, magnetic RAM, and ferro-electric RAM—that provide several benefits, including energy efficiency and tolerance of power failure. Flash memory has been used to augment traditional RAM in the Vista and Windows 7 "ready boost" feature, whereas the other two technologies are potential RAM replacements. Unfortunately, these nonvolatile memories allow information and attacks to *persist* indefinitely [Enck et al. 2008]. Interestingly, Microsoft has anticipated the security issues associated with persistent memory and designed the ready boost feature to encrypt all contents of flash, making it difficult for forensics investigators to recover useful data [Hayes and Qureshi 2009]. If these memories are adopted in future architectures, without adequate attention to encryption, there is the potential that memory-based attacks will become more prevalent.

In effect, FDE has pushed the vulnerabilities associated with persistent data stored on disk down into the next level of the memory hierarchy, which has proven equally vulnerable. The key concept by which vulnerabilities were mitigated on disk was encryption: encrypting the disk provided confidentiality preventing access to sensitive information. By migrating the same solution down into RAM, it will be possible to circumvent similar attacks at this lower level of the memory hierarchy.

The typical *threat model* assumed in the memory encryption literature involves hardware and/or software attack. Attackers are often assumed to have physical access to the vulnerable system where sensitive information can be captured in various ways. The primary goal of attackers is to steal secret information or code. Memory modification is sometimes discussed, although usually as a means to force a system to leak confidential information. Examples of these attackers range from those motivated by financial gain, such as bank employees capturing ATM pin numbers and criminals copying and distributing software (DRM) to those motivated by more nefarious goals such as reverse engineering or stealing intelligence from autonomous military vehicles.

Software attacks involve corrupt processes or the operating system itself. Since the operating system typically controls memory arbitration, it must either be trusted and considered part of the Trusted Computing Base (TCB) or dealt with in another way.

This is handled in different ways in the literature, with many adding a secure, trusted kernel to the list of assumptions of the work. Other approaches include only hardware in the TCB, treating the operating system as any other untrusted process. A hybrid approach includes some portion of a trusted kernel or a trusted hypervisor along with hardware support.

One of the main assumptions in the memory encryption literature is that the processor provides a natural boundary within which sensitive information can reside—it is a fundamental component of the TCB in most approaches. All components outside of the processor are assumed to be vulnerable to include RAM and its interconnections (data and address bus), other I/O devices, and so forth. Most schemes attempt to protect RAM and the data bus, and several also target the address bus [Duc and Keryell 2006; Dallas Semiconductor 1997], whereas other external components are not normally considered. A subset of the memory encryption literature additionally adds the cache-to-cache connections as a consideration when protecting multiprocessor systems.

Although the security of systems employing memory encryption is enhanced, attacks on the devices are still possible by etching away the chip walls with acid to reveal internal bus lines for microprobing, or electromagnetic and power analyses among other side channels [Ravi et al. 2004; Kocher et al. 1999]. For systems relying on *software-based encryption*, key expansion tables (e.g., Advanced Encryption Standard [AES]) are subject to cache attacks; a malicious process tracks and times cache accesses [Osvik et al. 2006]. The typical target of all of these attacks is the encryption key hidden within the chip boundary. Most of these approaches increase the attacker workload by an order of magnitude, require expert knowledge, and cannot be exploited remotely (excluding cache attacks) over a network [Suh et al. 2007]. Moreover, although tamper-resistant mechanisms are already available that significantly increase the barrier to entry [Chari et al. 1999], protecting circuits from invasive and side-channel attacks is an open research area that is not addressed in the main body of memory encryption literature. Protections such as FDE are equally available to criminals and well-intentioned users [Casey et al. 2011]. Disk encryption has been used to protect information on criminal activity and prevent successful prosecution. Some of the techniques identified to aid law enforcement (e.g., DMA-firewire attack) in the capture of key material on suspect machines would be thwarted by memory encryption—memory encryption could be used to further protect criminal activity. This article explores efforts to realize protection of confidentiality through memory encryption in the context of next-generation operating systems.

FULL MEMORY ENCRYPTION IN OPERATING SYSTEMS DESIGN

In general, encryption is used to provide four basic properties of protection: *confidentiality*, *integrity*, *authentication*, and *nonrepudiation*. In trusted computing and operating system security, these properties are realized through *authenticated booting*, ensuring that program code is not changed before it is loaded into memory, *memory authentication*, ensuring that program code is not changed during use, and *attestation*, ensuring that hardware and software have not been altered. Trusted software components, which make up part of the TCB, are booted and verified producing a *chain of trust*, without which the security mechanisms could be compromised before the system is initialized. Whereas a few of the works discuss the implementation of these other mechanisms, most assume that these components are functional and focus on the overhead of memory encryption in the steady state. Other important assumptions often include mechanisms for secure code delivery, key creation and escrow, interprocess communication, and I/O protection, among others. Memory authentication is often closely associated with memory encryption solutions; however, a thorough survey of memory authentication mechanisms is available [Elbaz et al. 2009].

Memory encryption is solely concerned with the *confidentiality* of data and code during execution, with the express purpose of increasing attacker workload associated with crafting exploits and stealing sensitive information. It is interesting to note, however, that memory encryption would also hamper attempts to inject code, generally assumed to require memory authentication. An adversary lacking an encryption key would be unable to successfully change an encrypted binary, as decryption would result in corrupt code and likely program termination [Barrantes et al. 2003]. Early work associated with Full Memory Encryption (FME) was dominated by the desire to provide DRM and particularly to prevent the theft of intellectual property associated with program source code. This is still the primary purpose in some systems (e.g., gaming systems); however, more recently, these techniques have become recognized as a method for removing vulnerabilities and protecting system users.

There are two general approaches to providing confidentiality with encryption that are commonly used in computer architectures based on *symmetric-* or *public*-key encryption techniques. Symmetric key encryption, based on a shared secret (key), is generally held to be more efficient (i.e., on the order of 1,000 times faster) but does not provide nonrepudiation and requires a nontrivial trusted key distribution scheme [Kaplan 2007]. Three common algorithms are typically used to realize this approach based on DES, Triple-DES, and AES. Public-key encryption involves the use of two interlocking keys, one held privately and the other published, from which all four properties of protection, including nonrepudiation, can be realized. This scheme has the advantage that public keys can be distributed across open networks. A broad variety of books are available that describe these core ideas; Mel and Baker [2001] is particularly accessible. In light of the speed and complexity involved in public-key encryption, it is unsurprising that the memory encryption literature typically uses symmetric key cryptography. However, delivery of encrypted code over the network may be facilitated using the public-key model [Kgil et al. 2005].

Unfortunately, computer users have consistently demonstrated an aversion to any form of increased response time, even when associated with increased security. Studies suggest that delays of longer than 150ms are perceptible to users [Muller et al. 2011]. FDE has only become viable because overheads have been reduced to acceptable levels. Achieving similar levels of acceptable performance for memory encryption offers a far more significant challenge: there is an existing, growing, and well-documented speed gap between processors and memory—improvements in processor speed are outpacing improvements in memory speed by an average of 18% per year [Hennessy and Patterson 2006]. Adding encryption latency to this already strained interface may require an overhaul of the basic fetch-decode-execute cycle employed by processors.

Added to the complexities of any memory encryption solution is the fact that, unlike the hard disk where data is sequentially stored for access, memory is used in a broad variety of dynamic access patterns. Numerous decisions must be made concerning the granularity of encryption in operating systems. For example, a running program will utilize RAM during execution for both stacks and heap space. The stack is accessed so frequently that adding encryption/decryption overhead to stack operations might prove prohibitive. Unfortunately, during context switches, registers containing sensitive information are normally saved to the stack in external memory. Additionally, the heap size, for any given program, is not normally known a priori. The complexities of memory mapped input-output peripherals result in an inability to cache mapped regions. This naturally presents a challenge if the overarching concept involves decrypting memory only after it is brought onto to the processor chip. It is not clear if the entire memory should be encrypted with a single key, or if shared libraries, individual programs, and/or data should be encrypted independently using separate keys. Alternatively, should individual functions or cache blocks be used as the unit of encryption?

All of these decisions incur a trade-off between the number of keys that must be securely stored versus the degree of protection and overlapping in operations that can be realized.

The literature on memory encryption is largely concerned with three core approaches based on hardware enhancements, operating system enhancements, and specialized industrial applications. These approaches are explored in the sections that follow. Unfortunately, almost all of the hardware and operating system enhancements have only been implemented through simulation or emulation, and as a result, the claims have yet to be validated and quantified on practical systems.

MONOLITHIC PROCESSOR ENHANCEMENTS

The general scope of hardware enhancements includes a number of approaches that have added specialized encryption units and/or key storage mechanisms to existing processor designs. In addition, several efforts have proposed inserting hardware into the system bus to leverage legacy code and hardware. Although the first patents detailing memory encryption were executed in 1979 [Best 1979, 1981, 1984], and the first paper detailing their use was published in 1980 [Best 1980], the body of in-depth academic research related to general-purpose memory encryption has occurred primarily in the past decade.

One of the earliest papers, often referenced by others of this genre, highlights an Execute-Only Memory (XOM) architecture [Lie et al. 2000]. This architecture was designed to combat software piracy and combines aspects of both public and symmetric key encryption. Public-key encryption is used to deliver binary code to the XOM chip, which maintains a unique private key. This allows vendors to encrypt the code for a particular system and ensures that it cannot be reused on another system. The header associated with the code includes a symmetric key embedded within it, used to segment memory into unique compartments at the granularity of a process. In order to map compartments to encryption keys, each compartment is tagged. A single null compartment is created to hold all unencrypted processes and libraries. This compartment enables communication between encrypted processes while allowing all processes to use shared libraries.

The XOM architecture assumes several hardware enhancements to existing processors. Special microcode is required to store the unique private key in a private on-chip memory. A symmetric-key encryption unit is added to the processor, together with a special privileged mode of operation for encryption. A hardware trap on instruction cache misses provides a segue into this encryption mode for encrypted code. When a cache miss occurs, the instruction is decrypted before being loaded into the processors instruction register. Although the authors state that encryption could be accomplished in software, they acknowledge that this would be very expensive in terms of overhead. Since many of the papers that follow XOM include similar hardware, only the differences or unique contributions of the other systems will be discussed.

XOM encrypts memory in a straightforward manner commonly known by the encryption community as *electronic codebook mode* but referred to in the literature as *direct encryption*. Each code block is decrypted after it is read from memory, by the encryption unit, and encrypted before it is written back to memory. Kgil et al. [2005] propose an additional chip enhancement targeted at improving the security of direct encryption, called *ChipLock*. This involves storing a small trusted part of an operating system kernel, called *TrustCode*, in a Read-Only Memory (ROM), termed *TrustROM*. Additional instructions are added to enable secure communication between the trusted and untrusted parts of the operating system. The TrustCode intercepts all system calls for memory access and performs encryption without the knowledge of the untrusted portion of the operating system. Symmetric keys are assigned at the granularity of the

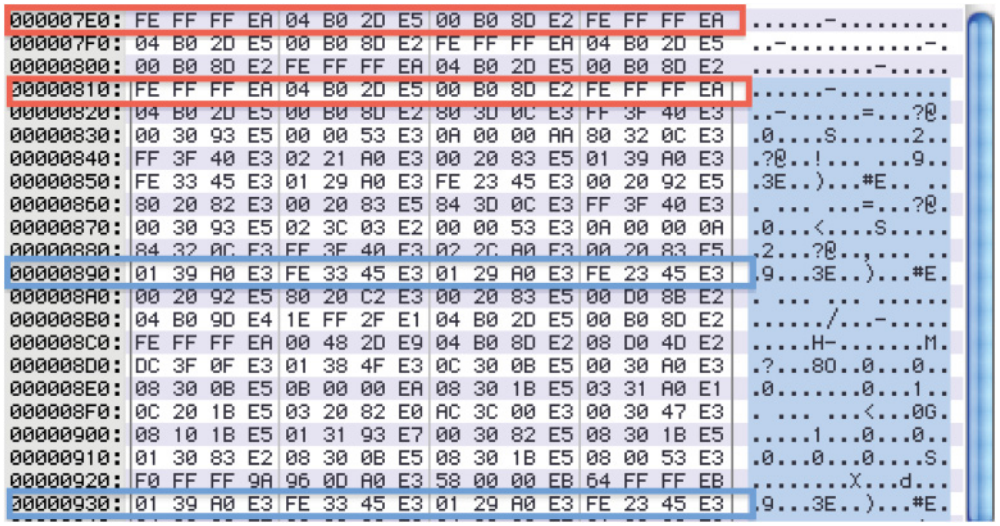

Fig. 2. Redundancies in 128-bit sections of a small selection of program binary code.

process as in XOM, with additional keys for shared libraries and the concept of a null bit for applications that are not encrypted.

Rogers et al. [2005] attempt to improve on direct encryption using an alternative mechanism, *prefetching*, which had already been improving the CPU-memory performance gap for decades. Prefetching uses stream buffers to capture *spatial locality* in programs by copying additional contiguous blocks of memory into local cache after each miss. These buffers are especially good at speeding up programs that exhibit *spatial locality* and *contiguous access*, such as scientific applications [Hennessy and Patterson 2006]. An alternative prefetching technique is also used that involves correlation tables to capture and reuse *temporal locality*—that is, complex and/or noncontiguous sequences of memory access.

In another direct encryption scheme, Hong et al. [2011] perform a trade-off analysis on the use of sensitive (encrypted) versus frequently accessed (unencrypted) data in embedded Scratch Pad Memories (SPMs). SPMs are software controlled SRAMs, as opposed to caches, which are typically controlled by hardware. There are numerous papers discussing both static and dynamic policies for SPM utilization to reduce power consumption and memory access latency. DynaPoMP was the first to consider partitioning the SPM into distinct areas with an area dedicated to sensitive code and data. The authors vary the size of the two partitions in an attempt to find the most efficient ratio. There is a common assumption that an encryption unit and special instructions are available in hardware.

Unfortunately, direct encryption schemes involve a one-to-one mapping between blocks of unencrypted and encrypted code. As a result, encrypted code portrays a similar statistical distribution as the unencrypted code, allowing a significant amount of information to be gleaned from frequency analysis [Chhabra et al. 2010]. Based on the typical AES encryption block size of 128 bits, programs tend to exhibit multiple redundancies that would lead to information leakage as shown in Figure 2.

After XOM, a number of papers attempt to mitigate this statistical weakness using a One-Time Pad (OTP) [Suh et al. 2003; Shi et al. 2004; Yang et al. 2005; Yan et al. 2006; Suh et al. 2007; Duc and Keryell 2006]. A traditional OTP is simply a source of random data that is used exactly once to encrypt a particular communication. This

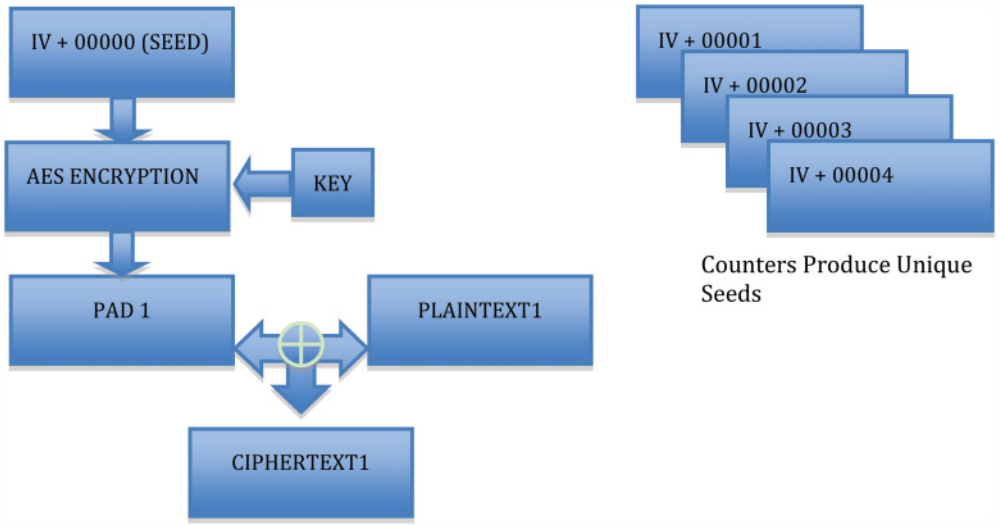

Fig. 3. Pseudo–one-time pad or counter-mode encryption.

is a form of symmetric-key cryptography since both the sender and receiver require the pad. Although variously referred to as Pseudo–One-Time Pads (POTPs) in the literature, this is more commonly known in the encryption community as counter-mode encryption. In computing, OTPs are created by encrypting a unique seed, typically producing a pad of 128 bits in length (i.e., the size of an AES encryption block) as shown in Figure 3. A fixed initialization vector (Nonce) is concatenated with a counter producing a unique seed. The seed is encrypted with a unique key generating the pad, which is then exclusively or'ed (XOR) with the plaintext to produce the cipher text. In memory encryption schemes, the counter is stored either internally, in a cached table that maps to a memory address, or unencrypted within the encrypted memory itself (i.e., RAM) since counter secrecy is not required [Yan et al. 2006]. When a memory reference occurs, the pad is regenerated, using the counter (and optionally some other component such as the virtual address) and initialization vector, then exclusively or'ed with the encrypted data to produce the original plaintext. Since the encryption operation no longer depends upon the data in memory, this regeneration can be overlapped with the memory read, decreasing the performance impact of decryption.

Although Aegis is an OTP approach, it was originally proposed as a direct encryption scheme in 2003. Suh et al. [2007] propose the OTP approach, perhaps illustrating the shift away from direct encryption in the community. One interesting contribution from this paper is the method of creating the unique key. The chip-specific encryption key is created by physically unclonable functions [Suh et al. 2003]. These functions make use of unique timing characteristics of "identical" models of the hardware to create the unique keys. Aegis is one of several approaches to include the idea of a small, protected security kernel that is separate from the rest of the untrusted operating system. Unfortunately, this kernel measures 74K lines of code for virtual memory management alone [Chhabra et al. 2011].

In Yang et al. [2005], the authors look to reduce the execution overhead of using OTPs by adding a sequence number cache (SNC) onto the chip below the L2 cache. Sequence numbers, in this paper, correspond to the counters used in Figure 3. However, the initialization vector is unique per cache block and corresponds to the virtual address. Since the addresses are unique across memory, the pads (and thus the ciphertext) will

be *spatially* unique. The counters are updated upon each write to memory ensuring *temporal* uniqueness (i.e., pads used for a single location will not be the same over time). The authors suggest that a reasonable addition to a chip would be an SNC of 64KB. Based on this limitation, two policies for using the SNC are described. In the first, only the portion of memory corresponding to the number of available sequence numbers stored in the SNC can be encrypted. The amount of protected memory is therefore limited by the SNC size. In the second method, additional memory lines are encrypted, and sequence numbers that do not fit in the cache are stored in plaintext in memory. Level-two cache is increased in both methods by 4% in order to store the virtual memory addresses used to index into the SNC, since only physical addresses are typically available above the level-one cache.

Yan et al. [2006] present *split counter-mode* encryption, in which they introduce major and minor page counters. In this scheme, a 4KB page has one 64-bit major counter and 64 7-bit minor counters (one per 64-byte cache line). Concatenating the page major counter with the cache line minor counter forms the overall counter. This counter is further concatenated with the memory block's virtual address and an initialization vector to form the unique seed. The vector can be unique per process, group of processes, or system based on security requirements.

In CryptoPage [Duc and Keryell 2006], the authors again attempt to enhance the OTP encryption scheme. In this case, they modify the Translation Look-aside Buffer (TLB) and page table structures, adding information for pad computation. Since the TLB and/or page table structures are always accessed before a memory read, the authors claim that the pad generation latency can be almost completely removed. This scheme is implemented on top of the *HIDE* memory obfuscation technique, whereby access patterns are permuted in memory at designated times [Zhuang et al. 2004].

In Address Independent Seed Encryption (AISE) [Rogers et al. 2007], the authors propose to use a *logical* identifier, rather than the virtual or physical block address, as the major counter portion of the seed. This scheme closely resembles split mode counters [Yan et al. 2006]. It is claimed that using an address independent seed enables common memory management techniques, such as virtual addressing, paging, and interprocess sharing. Chhabra et al. [2011] propose to build a secure hypervisor upon the AISE substrate. The hypervisor implements *memory cloaking*, whereby the operating system only has access to the encrypted pages of applications. The authors suggest that this cloaking will protect processes from vulnerabilities in the insecure underlying operating system, with an order of magnitude fewer lines of code than in Aegis.

Nagarajan et al. [2007] propose compiler-assisted memory encryption for embedded processors assuming some limited hardware support. They claim that the current counter-mode solutions require too much silicon space for small- and medium-size embedded processors. The compiler supports memory encryption by introducing special instructions to calculate OTPs prior to loads and stores, and assumes the existence of additional process-unique registers used to store the counters. Space for the unique key and global counter is also provided inside the CPU and the availability of a crypto unit is assumed. The compiler attempts to ensure that the counter used for a store is still available for successive loads from the same memory location. A global counter must be available for those loads and stores that do not match one of the process-unique counter registers. The authors claim that since frequently executed loads and stores exhibit highly accurate counter matching, 8 special hardware registers with 32 counters are sufficient for reasonable performance.

MULTIPROCESSOR ENHANCEMENTS

Chhabra et al. [2010] compare a Symmetric Multi-Processor (SMP) and a Distributed Shared Memory (DSM) design; they also provide a quick look at monolithic memory

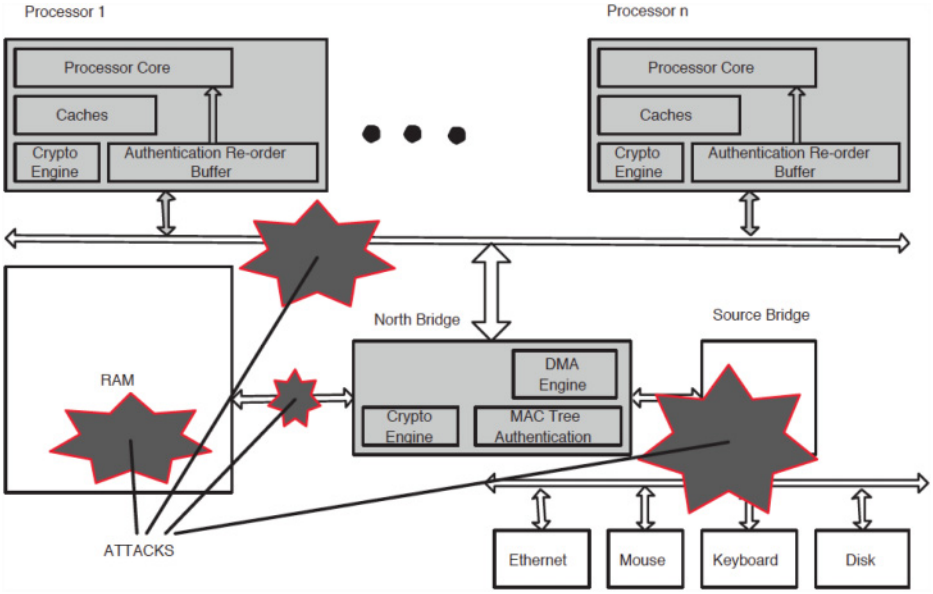

Fig. 4. SMP architecture with memory encryption support.

encryption. Whereas the efficiency of memory-to-cache confidentiality is the primary concern for monolithic processors, multiprocessor systems must also protect cache-to-cache traffic. In SMPs, the shared bus between caches and memory can be used as a way to coordinate messages between processors. This sharing is not available in DSM systems, which must use message passing. Additionally, DSM systems can be observed more easily than monolithic chips via interconnect wires that are exposed at the back of server racks [Rogers et al. 2008].

Shi et al. [2004] use OTP encryption both for memory-to-cache and cache-to-cache transfers, as shown in Figure 4. In this approach, sequence numbers (counters) are incremented in lockstep in each separate processor, resulting in a claim of "very low" overhead for cache-to-cache encryption. A hardware mechanism in the processors ensures that the sequence numbers begin differently after each reboot. Besides the typical crypto engines placed within each processor core, a separate crypto-unit is embedded in the north bridge memory controller for memory-to-cache transfers. For these transfers, 64-bit sequence numbers are stored in RAM, reducing the available memory by 25%.

SENSS [Zhang et al. 2005] utilizes OTPs for memory-to-cache transfers and AES cipher block chaining (CBC) mode for cache-to-cache transfers. This alternative to direct encryption divides the clear text into blocks and encrypts the first block with an initialization vector; subsequent blocks are chained together such that the output of the previous block is XOR'd with the input of the next before being encrypted. CBC implies sequential access since each block depends upon each previous block. RAM is typically accessed in a fairly random pattern, so this mode of operation is impractical except on a very small scale (e.g., per cache block). CBC is acceptable for cache-to-cache transfers, as only one previous encrypted block must be stored at each processor (i.e., there is no requirement for access to previously encrypted blocks). The authors propose a Secure Hardware Unit (SHU), located at each processor, comprising an encryption unit with associated storage for keeping track of communication. This storage includes memory for a group processor matrix and group information table. The group processor matrix is used by each SHU to determine if broadcast messages should be read. The

matrix is only 640 bytes in size, assuming a maximum of 32 processors. The information table contains the secret information for communicating between groups, such as the symmetric key and pads, and is estimated at 149KB. An additional 11 bus lines are used for control signals and to pass group i.d. numbers. In Jannepally Sohoni [2009], the SENSS scheme is improved using Galois Counter-Mode (GCM) AES, which provides both encryption and authentication simultaneously.

In I2SEMS, Lee et al. [2007] create a scheme that is claimed to be applicable to both SMP and DSM systems. They propose a global counter cache that assigns different sections of the overall counter space to processors (akin to assigning blocks of IP addresses to groups of computers). The blocks of counters are also broadcast to all processors so that they can begin precomputation of pads. Each processor has a keystream (pad) queue, keystream cache, and keystream pool. The queue and cache both contain pads for encryption. The queue has new pads, whereas the cache contains pads that have been used previously. The authors claim that pads may be reused as long as the plaintext has not been modified and that their scheme scales well to large numbers of processors since more than 25% of pads are reused. The keystream pool holds pads for incoming data; the pads are chosen based on prediction with the aid of the broadcast scheme.

The first paper to exclusively address DSM systems was by Rogers et al. [2006], who again make use of counter-mode encryption. Since the memory-to-cache scheme is similar to those already discussed, we only focus on the cache-to-cache scheme. The authors propose three methods for managing the pad counters: *private*, *shared*, and *cached* counter stream. In the first *private* method, tables are kept within each processor with separate counters for send and receive operations to/from every other processor in the system. Although this technique allows for nearly perfect pad hit rates, and therefore very low overhead, it suffers from large storage needs (180KB in each processor for a 1,024-processor DSM). The second *shared* scheme aims to reduce the storage requirement by eliminating half of the table. Instead of keeping track of send counters for each processor, only one counter is kept for sending pads. This results in increased execution overhead, since messages are less likely to arrive contiguously and therefore must be recomputed. The final *cached* scheme takes advantage of the intuition that processors in DSM systems often communicate in cliques [Lee et al. 2007]. The overall table size is thus reduced to a quarter of the private scheme's memory with minimal impact on execution overhead. In a subsequent paper, Rogers et al. [2008] identify the previous scheme as a two-level approach, since remote memory requests will first be decrypted by the owning processor and then re-encrypted for cache-to-cache transmission to another processor. In the new scheme, a single mechanism is used for both memory-to-cache and cache-to-cache transfers bypassing the unnecessary decryption and re-encryption. The associated hardware includes a 32-entry buffer (1KB) for counter prediction and a 32-entry mask buffer that stores a bit vector of recent data block accesses (512 bytes).

BUS INSERTS

Another area of active research involves placing specialized encryption hardware outside of the CPU. The locations include the memory bus (i.e., externally between system memory and the CPU) and within RAM. The primary goal of this approach is to increase the likelihood that this solution will be adopted, since re-engineering of commodity processors is not required. One such approach, SecBus [Su et al. 2009a] shown in Figure 5, can be located at the front end of the memory controller. The authors state that this method of modification is required in many user markets when embedding new functionality into systems with legacy CPUs. SecBus is essentially a cryptographic coprocessor with internal storage and bus manager. The Page Security Parameters

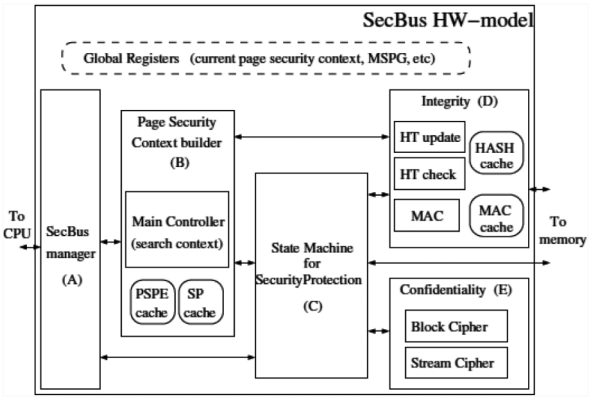

Fig. 5. SecBus hardware augmentation model.

Entry (PSPE) includes information to map pages to corresponding Security Policy (SP), which includes a confidentiality mode, integrity mode, and secret key. SecBus includes the ability to choose between multiple encryption modes based on the type of memory (i.e., code or data).

Enck et al. [2008] designed a Memory Encryption Control Unit (MECU) to again be placed on the memory bus between the processor and RAM. The goal of MECU is to provide the same guarantees of security provided by the volatility of traditional RAM when utilizing nonvolatile main memory. MECU uses an OTP scheme with internal storage for the array of counter seeds and the encryption engine. A secret key and master counter, which tracks the greatest overall counter, are stored on a removable smart card. In order to reduce the storage requirement, the encryption chunk granularity is increased from one cache line to n, where n is 256 in the common case but can grow to the entire memory for experimentation.

With the same goal as Enck et al. [2008], Chhabra et al. [2011] propose placing the cryptographic engine and other required hardware in nonvolatile RAM modules. Their scheme keeps most of the RAM encrypted with a smaller group of frequently accessed pages in plaintext in a similar fashion to Hong et al. [2011]. The authors claim that by doing this, the remainder of the RAM can be encrypted at power-down within 5 seconds, paralleling traditional RAM volatility.

OPERATING SYSTEM ENHANCEMENTS

Similar to the bus insert method for enabling memory encryption, software-only approaches seek to provide solutions that can be implemented without major changes to applications or commodity hardware to increase the likelihood of adoption.

Chen et al. [2008] propose an operating system controlled memory bus encryption technique for systems that offer SPMs or cache locking that is software controllable. Both types of memory are available in some embedded processors including the Intel XScale series. A new symmetric key is generated each time the system is booted, and random vectors (32 bits generated using /dev/urandom and padded with 0's) are used to initialize AES encryption at the granularity of a page. The vectors are then placed in memory with the pages. This scheme requires a 0.4% space overhead when used with 1KB pages. When a page fault occurs for a secure process, a specially crafted handler moves the encrypted page into the chip boundary and decrypts it there, placing it into the cache, which is then locked to prevent leakage of sensitive data. The locked region holds several pages of data and encryption variables. In order to facilitate

this special handling, a Boolean status variable is added to each process descriptor residing in kernel address space. The authors note that the scheme is appropriate when embedded systems designers can tolerate a significant performance overhead for protected processes.

In Cryptkeeper, Peterson [2010] modifies the virtual memory manager and partitions RAM into two parts: the plaintext *Clear* and the encrypted *Crypt*. Essentially, this technique aims to reduce the amount of sensitive data available at any time in memory. All pages initially start in the clear, and the number of Free Clear Pages (FCP) is reduced with each allocation. The least recently used pages are encrypted and moved to the Crypt when the limit of FCP runs low. This operates under the assumption that the number of high-use pages will be small, and therefore most infrequently used pages will be encrypted. This has the unfortunate side effect of maintaining all of the important pages in the clear. A prototype Cryptkeeper system was designed based on the Linux 2.6.24 kernel. The kernel page structure was extended to include information indicating whether a page is in the Clear or Crypt portions of memory.

SPECIALIZED INDUSTRIAL DEVICES

Industry offers several solutions for memory encryption, including low-frequency specialized processors for ATM use, expensive tamper-resistant coprocessors for financial transactions, proprietary gaming systems, and, more recently, enabling technologies in commodity processors to enhance trust.

The Dallas Semiconductor 5002FP secure processor is an 8051-compliant processor and runs at a maximum frequency of 16MHz [Dallas Semiconductor 1997]. The processor encrypts memory addresses to prevent traffic analysis on the memory bus in addition to data. The device uses spare processor cycles to place dummy memory accesses on the bus since analysis of memory access patterns can reveal useful information (e.g., encryption keys or sensitive algorithms) to attackers [Gao et al. 2006]. All external memory is encrypted via a proprietary encryption algorithm with a 64-bit secret key that is stored in a tamper-protected, battery-maintained static RAM. Plaintext code is uploaded via serial port, and a firmware monitor encrypts it and stores it in external RAM. The 5002FP is commonly used in credit card (i.e., point of sale) terminals, automated teller machines, and pay-TV decoders [Yang et al. 2005]. A newer version (DS5250) includes a larger 1KB instruction cache, which, according to Dallas Semiconductor, reduces the effect of memory encryption on execution speed, providing a 2.5X performance improvement. The newer processor runs at a maximum frequency of 25MHz.

Another active area of secure hardware used in industry is the cryptographic coprocessor, such as the IBM PCI-4758. These coprocessors include an impressive array of technology, including a secure processing environment, microprocessor, custom encryption and random number generation hardware, and shields and sensors (to help protect against destructive attacks) [Howgrave-Graham et al. 2001]. However, they are generally limited to IBM server platforms under customized contracts and tend to be used for financial and banking systems. A modified version of CP/Q message-passing kernel runs on the system, providing a subset of typical features. The secure module is encased in a flexible mesh of overlapping conductive lines meant to prevent any physical intrusion. If such intrusion is detected, the system responds by zeroizing the internal RAM, which holds the secret key. The stated purpose of the IBM secure coprocessor is to offload computationally intensive cryptographic processes (e.g., specialized financial transactions) from the host server.

Although mostly constrained for use in playing games and other entertainment media (unless compromised), gaming systems are some of the most capable (e.g., fast processor speed and relatively large storage) to incorporate memory encryption

techniques. As an example of these systems, the Xbox 360 provides encrypted/signed bootup and executables, partially encrypted RAM, and an encrypted hypervisor [Steil and Domke 2008]. These mechanisms are provided via a Microsoft proprietary processor with 64KB of internal RAM, random number generation, and encryption as opposed to the "off the shelf" processor used in the original Xbox. Although it is possible to use the Xbox as a general-purpose platform, this requires compromising the system's security measures first. Alternatively, the Sony Playstation 3 includes many of the same security mechanisms of the Xbox 360 but allows the end user to partition the hard drive for use with a chosen (e.g., Linux) operating system. However, the proprietary security mechanisms of the Playstation 3 are not available to the additional operating system [Conrad et al. 2010].

The Trusted Computing Group (TCG) designed the Trusted Platform Module (TPM) based on the IBM 4758 secure coprocessor [Vandana 2008]. The TPM provides secure key storage and the capability for platform measurements for chain-of-trust booting. The current specification for the TPM calls for it to be attached to a typical motherboard via the Low Pin Count (LPC) bus. The TPM provides nonvolatile storage for encryption keys and an encryption engine including support for RSA, SHA-1 hashing, and random number generation. The LPC bus is limited in speed, and the cryptographic engine on the TPM is not meant to be a cryptographic accelerator. More than 350 million TPMs were deployed as of 2010 and can be found in many laptops and general-purpose computers (disabled by default) [Dunn et al. 2011]. On its own, the TPM would not be powerful enough to provide general memory encryption with acceptable overhead. However, the TPM may be used to provide secure key storage between power cycles. Unfortunately, a small weakness still exists in that keys must be sent in the clear over the LPC bus to the CPU, allowing a bus snooping attack to capture them [Simmons 2011]. Other interesting methods to store encryption keys have been described recently in schemes targeted at preventing cold-boot attacks on FDE. For example, Muller et al. [2011] describe TRESOR, a technique for utilizing CPU debug registers for encryption key storage. In order to protect against memory attacks on the key, the decryption routines are carefully written in assembly to avoid using the stack, heap, or data segment during decryption. By utilizing Advanced Encryption Standard–New Instructions (AES-NI), TRESOR was shown to perform better than software-based FDE (17.04MB/s vs. 14.67MB/s) with the additional protection. A similar approach is taken in Simmons [2011] except that registers used for performance counting are targeted for master key storage with multiple encrypted keys being stored in RAM.

Intel has recently filed several patents for processors incorporating memory encryption, perhaps indicating a move toward support in commodity processors [Gueron et al. 2012; Gueron et al. 2013]. The patents describe a new processor with hardware including a memory encryption engine and on-chip storage for counters. The hardware described in the application modifies the AES-XTS *tweak* mode of operation. XTS stands for XEX-based tweaked codebook mode with ciphertext stealing, and this mode is typically used for disk encryption [Martin 2010]. A tweak is similar to an initialization vector and is an additional input to a cipher designed to protect against similarities in ciphertext. For disk encryption, the tweak tends to be the sector number. In Intel's patents, the tweak is extended to include a time stamp or counter value along with the memory address. The counter is updated each time a cache line is written, providing protection against a replay attack where a chunk of memory is copied and inserted back into memory at a later time.

COMMODITIZED SECURITY HARDWARE

Most of the approaches in the memory encryption literature assume that several components are necessary for secure, efficient performance: a way to generate and securely

store encryption keys (i.e., not in RAM), and hardware to accelerate encryption performance. Although not targeted specifically at memory encryption, nascent technology could be used to form the basis of an encrypted memory solution for general-purpose systems. One of the developers of IBM's 4758 cryptographic coprocessor has suggested, for example, that a general-purpose system with hardware support (e.g., a TPM) could theoretically be turned into a somewhat less secure but more pervasive and less expensive version of the 4758 [Smith 2004]. Encryption engines have been added to Intel's core i5 and i7, AMD's bulldozer, and various embedded processors [Muller et al. 2011]. For X86 systems, Intel's AES-NI includes six instructions to speed up key expansion and encryption. Intel states that the new instructions can provide a two- to three-time performance improvement over software-only approaches for nonparallel modes of operation such as CBC encryption [Gueron 2010]. Further, a 10-fold improvement can be realized for parallelizable modes including CBC-decrypt and counter mode encryption. As an example of the performance improvements possible, the authors ran TrueCrypt's encryption algorithm benchmark test on a MacBook Pro with an Intel i7 dual-core, 266GHz CPU. Using a 5MB buffer in RAM, the throughput averages 202MB/s without AES-NI support, and 1GB/s with it—approaching the speed required to overcome encryption overheads on general-purpose systems.

Henson and Taylor [2013a,b] are among the first to take advantage of this commoditization of security hardware for use in implementing memory encryption. The IMX53 development board is used in conjunction with an ARM cortex A8 processor that contains security hardware within its boundary. The hardware consists of encryption and hashing engines and random number generation as well as facilities for trusted booting. A small (∼35KB) microkernel called *Bear* is developed and integrated with the A8 and security hardware. As the work is implemented on commodity hardware, it considers many of the details that are not thoroughly addressed in the other surveyed literature (i.e., simulation work). For example, encryption is explored at process component granularity (e.g., stack, heap, code) with analysis of the overhead for encrypting each component. The work takes advantage of on-chip, Internal RAM (iRAM) as well as cache to provide the secure processing environment. Outside of this chip boundary, all code and data are encrypted. Most of the memory encryption functionality is tied to the context-switching routines in the microkernel. The microkernel fits into the iRAM and is part of the TCB in this work.

ANALYSIS

Although the primary goal of memory encryption architectures is security, the work tends to focus on the overheads involved, both in chip area and performance degradation. This is unfortunate although unsurprising given that most of the work is simulated, and it is within the intricacies of implementation that security vulnerabilities tend to be found. The analysis here focuses on the data available, including encryption latencies, performance degradation, simulation environments, operating system assumptions, overall space requirements, user requirements, and general observations regarding security.

Since the performance degradation of memory encryption results in less likelihood of its use, it is an extremely important factor in the comparison of different schemes. One of the major issues with the body of literature is the lack of a common set of measurement standards, with explicit assumptions regarding memory access latency, encryption latency, and so forth. This makes it difficult to directly compare approaches and draw valid conclusions. Encryption latencies are typically given as the number of cycles required to encrypt/decrypt a cache line that varies from 16 to 128 bytes, typically using a value of 64 bytes. The latencies range from 11 to 160 cycles, with 80 being the most common value (especially in the multiprocessor work). Rogers et al. [2006]

state that 80-cycle latency is assumed in order not to penalize the direct encryption scheme (upon which they are trying to improve) since a recent (circa 2006) hardware implementation required more than 300ns. Cycles and nanoseconds are often used interchangeably, since many of the systems modeled are based on 1GHz processors. Low encryption latencies are possible, although at the cost of large die area, making them appropriate for powerful processors. For example, it is claimed in Suh et al. [2003] that 40 cycle-latency is achievable with four AES units chained together requiring 300,000 gates. In Aegis [Suh et al. 2007], a single AES unit is estimated at 86,655 gates, which the authors claim is modest when compared to the size of commercial cores. Unfortunately, the OR1200 soft core used to demonstrate Aegis is only approximately 60,000 gates (meaning that one AES unit is 144% of the original core size).

The methods used for determining performance include mathematical models, simulation, kernel prototypes, and FPGA prototypes, with various benchmarking suites used in the latter three. Simulation is performed with (in order of decreasing usage) SimpleScalar, Simics, SESC, GEMS, SOC designer, RSIM, and M5. Benchmark suites used include SPEC2000, SPLASH2, Mediabench, EEMBC, and several user-developed varieties such as one entitled "memeater." A group of the simulations utilize SimpleScalar, and Duc and Keyell [2006] note that this simulator neglects the impact of the operating system and other running processes. Besides these limitations, some authors admit a lack of model fidelity with significant differences between systems modeled and those targeted. For example, in Chen et al. [2008], an $\times 86$ architecture is modeled since it happens to be better supported by the simulation tool (Simics) even though the scheme is actually targeted for embedded-ARM systems. Unfortunately, even if a system under test were to be modeled perfectly, the simulation tools themselves have been shown to sometimes exhibit behavior unlike real systems. In [Muller et al. [2011], the behavior of CPU registers is interrogated under simulation in QEMU with the contents surviving soft-boot. This behavior would circumvent the protections afforded in that work; however, real hardware behaves differently and zeroes out the registers.

A summary of the featured techniques is presented in Table I to provide an overview of memory encryption. The table includes basic characteristics of each approach, such as complexity information including execution and storage overheads. In order to fairly compare the different schemes, several assumptions were made. For example, the size of internal storage required is sometimes dependent on the size of RAM, and where possible, an assumption of 1GB is made. Similarly, an assumption of 32 processors is made where possible for the multiprocessor approaches. When there is no data available, an element of the table is left blank. Two values are commonly reported in the literature with regard to execution overhead: worst case (max) and the average (based on some suite of benchmark tests) percentage slowdown when compared to nonprotected execution. Storage overheads typically break down into internal (cache) and external (RAM) usage (and one example of the increase to overall code size). Operating system approach indicates whether the authors assumed the existence of a secure kernel (A), described hardware to protect the processes from an insecure kernel (H), or ignored the operating system (I) (further discussion of this requirement follows). Finally, slightly fewer than two-thirds of the authors included memory integrity (I) along with memory confidentiality (C) mechanisms. Where possible, results (e.g., execution overhead and storage) are provided for memory encryption only. Maturity indicates how the technique was evaluated if not a commercial product. Methods appear in the table as they are presented in the survey and detailed in the approach column: monolithic processor, multiprocessor, bus insert, or software/direct or counter-mode encryption.

Security level refers to the overall security of the memory encryption approach with the following factors from the table taken into consideration: category, operating system

Table I. Summary of Memory Encryption Techniques

Reference	Category	Execution Overhead Max/ Average %	Storage Overhead (I)nternal, (R)AM, (C)ode	Operating System Approach	Maturity	(C)onf (I)ntegrity	Encryption Algorithm	Full/Partial Memory Encryption	*Security Level 1–6*
Lie et al. [2000]	Mono/Direct	50 /	I – Private Mem & XVMM	H + Virt Machine	Math	C + I	3 DES	PME	*3*
Kgil et al. [2005]	Mono/Direct	/ 3	I – Key Table	H + Part of Kernel	Sim	C + I	AES	FME	*6*
Rogers et al. [2005]	Mono/Direct	/ 1	1.5 MB-I	I	Sim	C	UNK	FME	*3*
Suh et al. [2003]	Mono/Counter	32 / 4.5	12 KB-I 6%-R	H + Part of Kernel	P-FPGA	C + I	AES	FME	*6*
Yang et al. [2005]	Mono/Counter	/ 3.9	64 KB-I 1.5%-R	H	Sim	C	AES	FME	*6*
Yan et al. [2006]	Mono/Counter	9 / 2	32 KB-I 1.5%-R	A	Sim	C + I	AES-GCM	FME	*5*
Duc and Keryell [2006]	Mono/Counter	7.4 / 3		H	Sim	C + I	AES	PME	*5*
Nagarajan et al. [2007]	Mono/Counter	/ 2.3	32 B-I 12.5%-R 3%-C	I	Sim	C	AES	FME	*4*
Chhabra et al. [2011]	Multi/Counter	13 / 5.2	32 KB-I 1.6%-R	H	Sim	C + I	AES	FME	*6*
Rogers et al. [2007]	Multi/Counter	13 / 1.6	32 KB-I 1.6%-R	I	Sim	C + I	AES	FME	*4*
Shi et al. [2004]	Multi/Counter	55 /	32 KB-I 25%-R	A	Sim	C + I	AES	FME	*5*
Zhang et al. [2005]	Multi/Counter	/ 12	149 KB-I	H	Sim	C + I	AES-CBC	FME	*6*
Jannepally et al. [2009]	Multi/Counter	/ 5.2	4.8 KB-I	I	Sim	C + I	AES-GCM	FME	*4*

(Continued)

Table I. Continued

Reference	Category	Execution Overhead Max/ Average %	Storage Overhead (I)nternal, (R)AM, (C)ode	Operating System Approach	Maturity	(C)onf (I)ntegrity	Encryption Algorithm	Full/Partial Memory Encryption	*Security Level 1–6*
Lee et al. [2007]	Multi/Counter	10 / 4	608 KB-I	I	Sim	C + I	AES-GCM	FME	*4*
Rogers et al. [2006]	Multi/Counter	/ 6	4.5 KB-I	I	Sim	C + I	AES	FME	*4*
Rogers et al. [2008]	Multi/Counter	7 / 1.6	33.5 KB-I	I	Sim	C + I	AES	FME	*4*
Su et al. [2009a]	Insert/Direct	23,753/ 100		A	Sim	C + I	UNK	FME	*3*
Enck et al. [2008]	Insert/Counter	4.4 / 2.1	131 KB-I	H	Sim	C	UNK	FME	*4*
Chhabra and Solihin [2011]	Insert/Direct	20 / 7.2	78 MB-I	H	Sim	C	AES	PME	*4*
Chen et al. [2008]	Soft/Direct	78 / 37	0.4%-R	A	Sim	C	AES-CBC	PME	*3*
Peterson [2010]	Soft/Direct	800 / 9		I	P-Soft	C	AES-ECB	PME	*2*
Henson and Taylor [2013b]	Mono/Direct	50 /		Microkernel	P-Hard	C	AES	FME	*6*

approach, encryption algorithm, and partial versus full memory encryption. Specifically, the five sections are scored with maximum points as follows: category (1), operating system approach (2), encryption algorithm (2), and encryption level (1). A score of 6 represents a system capable of addressing a wider range of memory threats than those with lower scores. For category, no points are given for bus inserts and software approaches due to inherent weaknesses of these techniques when compared to hardware approaches. The operating system approach is scored as follows: hardware (2), assumption of secure operating system (1), no discussion (0). The encryption algorithm used receives 2 points for AES and 1 point for DES or unknown algorithms. We will consider partial/full memory encryption and security level in more detail. Although partial memory encryption schemes are typically used to decrease both space and execution overheads, they place the onus for identifying secure components, a nontrivial task, on application or system designers. Today, an analog can be observed in the adoption of hard disk encryption technologies, whereby administrators struggling to identify which files (or parts of files) require encryption are opting instead for FDE [Brink 2009]. Since it is difficult for end users to properly determine which processes should be encrypted, partial memory encryption receives 0 points, with FME receiving 1.

A comparative analysis on the relative security of these techniques is nontrivial, and it is important to note that the analysis in this work favors approaches that aim to mitigate a wide range of threats over those with a narrower scope. For example, FME will receive a higher score than an approach to add volatility to magnetic RAM, making it behave more like traditional RAM. Additional factors to consider when analyzing these works include consideration of implementation details outside of the "steady state" such as key escrow, delivery of secure code, interprocess communication, and so forth, although these are not used for the purposes of scoring.

Each approach is qualitatively evaluated on the five components listed earlier, receiving a total score ranging from 1 to 6. As an example, the Aegis approach [Suh et al. 2003] is among the highest security level of the works surveyed (6): the category is monolithic processor with encryption support built in (+1); the operating system approach includes both hardware and a small, trusted kernel (+2); the AES encryption algorithm is used (+2); and FME is provided (+1). Although not part of the score, much of the additional details required for a fully functional, secure implementation are discussed in Aegis. It is unsurprising that the approach with the highest security evaluation is also among the most mature (implemented as an FPGA prototype) since implementation allows for exploration of security trade-offs. In contrast, operating system controlled memory encryption [Chen et al. 2008] is classified among the lowest security levels (3): this approach is software based (+0); *assumes* that the kernel is secure (+1); utilizes AES encryption (+2); and targets partial memory encryption (+0). Additionally, this work lacks sufficient detail for a fully functional system and assumes that the attacker is a *clever outsider*.

For direct encryption, the performance overhead ranges from a claimed low of 1% in Rogers et al. [2005] based on simulation of predecryption to a high of 50% for XOM [Lie et al. 2000] using mathematical analysis based on a worst-case scenario. Rogers et al. find an average slowdown for a model of XOM of 21% based on the same 18 SPEC2000 benchmarks used in their own simulation work. In four particular benchmarks (applu, bt, ft, and mcf), the overall execution time for predecryption is similar to the direct encryption scheme because prefetching adds mispredicted memory references to bus traffic, increasing contention. Overhead for OTP-based encryption, in monolithic chips, ranges from a claimed 1.6% for AISE (SESC and 21 CPU2000 benchmarks) [Rogers et al. 2007] to up to 50% for the basic model in CryptoPage (SimpleScalar and 10 CPU2000 benchmarks) [Duc and Keryell 2006]. The authors of CryptoPage claim that only 1% of this overhead is attributable to the memory encryption.

For multiprocessor systems, the reported overheads range from a low of 4% in I2SEMS (Simics + GEMS and 4 SPLASH2 benchmarks) [Lee et al. 2007] to a high of 55% in Shi et al. [2004] (RSIM and 6 SPLASH2 benchmarks). I2SEMS is claimed to work equally well on both SMP and DSM systems, but the simulation environment is limited to SMP. Cache-to-cache overheads are very low (especially for SMP systems that use the shared bus for synchronization) in these multiprocessor schemes. All of the multiprocessor schemes build upon work in the monolithic memory encryption area and use the counter- mode (OTP) model.

There are only two models surveyed for hardware insert, and they exhibit very different performance characteristics. MECU [Enck et al. 2008] is based on the OTP scheme and exhibits 2.1% and 4.1% overhead based on block sizes of 256 and 4,096 cache lines, respectively, and SimpleScalar simulation with 5 SPEC2000 benchmarks. SecBus [Su et al. 2009a] is based on direct encryption and exhibits worst-case slowdowns of 472% based on various EEMBC benchmarks and SoC designer. Besides the method of encryption, the architectures modeled add to the significant differences in overhead. Whereas SecBus is simulated on an embedded system with 16KB L1 cache and no L2 cache, MECU is modeled after an x86 system with 32KB L1 and 256KB unified L2. Clearly, the amount of cache available has a huge impact on performance. If complete working sets fit into a system's cache, the penalty for memory encryption includes only the initial decryption time, which is amortized across the entire duration of the process.

As might be expected, the software-only approaches suffer from impractical overheads. Chen et al. [2008] simulated operating system controlled memory encryption and report from 137% to 850% overhead based on Simics and Mediabench benchmarks. In Cryptkeeper [Peterson 2010], the overhead to read a page when compared to an unprotected system is 6,015%. As far as commercial hardware, there is no literature available reporting the performance degradation of either the Dallas Semiconductor chips or the IBM cryptographic coprocessors (e.g., PCIXCC). However, these solutions run at slow overall frequencies (25MHz and 266MHz, respectively) and are not particularly well suited for general-purpose systems. The IBM PCIXCC coprocessor has a reported AES-128 throughput of 185MB/s.

In general, the counter-mode methods exhibit less computational overhead than the direct encryption techniques and are resistant to direct encryption's statistical weaknesses. However, the choice of size for the counter is critical since a "wraparound," whereby the counter resets to zero, requires a change of key in order that each pad is only used once (a condition necessary to ensure protection from chosen plaintext attacks) [Lipman et al. 2000]. In the case where only one key is used, the entire memory then requires re-encryption. This re-encryption can be costly depending on the size of memory and results in a temporary freezing of the system, which is unacceptable for real-time performance [Yan et al. 2006]. Choosing a value too small will result in too many re-encryptions, but choosing one too large will require unacceptable amounts of storage space either in cache or memory. For example, in Suh et al. [2003], the authors suggest that 32 bits is an appropriate size for the counter. However, even at this size, and based on their simulations, a re-encryption is required every 5.35 hours on average and every 35 minutes for a particularly memory-intensive program. In Yang et al. [2005], the authors choose to disregard the problem since the provided security is assumed to be no weaker than that of the XOM scheme, whereas the wraparound issue is not considered at all in Suh et al. [2007]. Yan et al. [2006] attempt to address the counter size versus re-encryption problem with their split-counter encryption scheme. With larger page counters and multiple smaller per-memory block counters, overruns result in a much finer granularity of re-encryption (per page instead of per process). Since some pages are written back to memory more often than others, the overall

necessity for re-encryption is reduced, since the fastest incrementing counter would have controlled the entire memory space in previous schemes. Another critical decision involves where to store the counters.

Although using cache is obviously faster, it is also problematic because cache resources are typically limited and expensive. If pre-existing cache space is utilized instead, additional memory references occur since part of processes' working sets are forced out of cache (essentially reducing the size of the usable cache, causing capacity misses). For example, in Yang et al. [2005], the authors state that a 1GB memory space would require more than 8 million sequence numbers based on cache line granularity and a cache line size of 128 bytes. Adding a cache that large (∼28MB) is unreasonable, so the authors suggest adding a much smaller 64KB one. However, this design decision either limits the security of the system, since a large part of memory would be unencrypted, or some sequence numbers would be stored in memory. There are 32K numbers (2 bytes each) stored in the SNC covering 32K L2 cache lines and 4MB of memory. Although RAM is slower than cache, the seed (which is smaller than a cache line) is the first memory access and would arrive earlier than the rest of the reference. Although this does not hide as much latency as using cache, it is an improvement over the direct encryption scheme. This technique would also render part of RAM unusable, as it would be utilized for additional storage.

In AISE [Rogers et al. 2007], the authors suggest that all of the previous OTP schemes are flawed in their use of memory address as part of pad computation. Using virtual addresses as a component of the input to the pad seeds may lead to a vulnerability, since separate processes will use the same address tweak as part of the seed (breaking the requirement for pad uniqueness). Additionally, using the virtual address for pad computation can cause problems for shared memory interprocess communication, since the pads would be different for the various processes even though both need to access the plaintext. For schemes using the physical address as part of the pad computation, there are other issues when swapping to the backing store. Since pages in memory that are swapped out are likely to reside at a new physical address when brought back in, there is a potential for pad reuse or the requirement for a decryption and re-encryption of a page loaded into a different address.

Industrial implementations have been shown to be vulnerable to attack. Kuhn [1998] demonstrates what is essentially a brute-force attack on the 5002FP. External hardware is used to control input to the processor and force it to power cycle. After each power-on, different encrypted "guesses" (possible instructions) are fed to the system, and the output ports are observed. The 5002FP had been described as the most secure processor available for commercial users at the time of this successful attack, which required a personal computer and a device built in a student laboratory for about $300. One of the reasons the 5002FP is vulnerable to brute-force attack is the small size of the plaintext. Kuhn notes that encryption performed over whole cache lines (of at least 8 bytes) instead of on single bytes would make the brute-force attack impractical. There is no known example of a successful attack against the IBM cryptographic coprocessors. However, these coprocessors tend to be used for highly specialized applications and are difficult to upgrade [Suh et al. 2007], making them undesirable for general-purpose computing environments. In fact, one of the designers of the IBM-4758 has noted his frustration with their *expense* and *modest processing environment* [Smith 2004; Gutmann 2000]. Although the TPM chip has been included in various trusted computing schemes, it is potentially vulnerable to the same types of snooping and bus injection attacks used against systems with unencrypted memory [Shi et al. 2004; Suh et al. 2007; Simmons 2011]. In fact, when utilizing the TPM with bitlocker drive encryption, the secret key is copied into RAM, making it vulnerable to capture via

cold-boot and other attacks as demonstrated in Halderman et al. [2008]. Since the key must be in RAM for bitlocker to function properly, the additional protection of the TPM is potentially nullified.

There are three basic approaches in the literature surveyed with regard to operating systems. The problem lies in the fact that without a secure (trusted) operating system, extra protections must be placed in hardware to prevent a compromised system from breaking the confidentiality of other processes. For example, when processes are context switched by the operating system, the registers and other internal memory will be in plaintext. The first approach is to explicitly assume the existence of a secure operating system [Chen et al. 2008; Shi et al. 2004; Yan et al. 2006; Suh et al. 2003; Su et al. 2009b; Chen and Morris 2003]. Some of the papers taking this first approach discuss implementation requirements, but none have been developed. In the second approach, the complexity of the hardware is increased in order to protect all processes (including the operating system) from each other [Kgil et al. 2005; Yang et al. 2005; Duc and Keryell 2006; Enck et al. 2008; Lie et al. 2000; Platte et al. 2006; Zhang et al. 2005; Chhabra et al. 2011]. One example of such hardware includes special instructions and extra registers that are called before context switches [Lie et al. 2000]. The internal registers are then encrypted strictly by the hardware before the kernel can intervene and complete the context switch as normal. Although several papers note the importance of working on a secure kernel to complement secure architectures, we have found no work to date suggesting the completion of any such effort. In the third approach, the requirement for a secure operating system is simply not addressed [Nagarajan et al. 2007; Rogers et al. 2005; Rogers et al. 2007; Hong et al. 2011; Lee et al. 2007; Jannepally and Sohoni 2009; Rogers et al. 2006; Rogers et al. 2008].

CONCLUSION

This survey has considered the research challenges associated with FME and distinguished three primary groups of techniques that attempt to solve those challenges—hardware enhancements, operating system enhancements, and specialized industrial devices. Although the concept of memory encryption has existed for more than three decades, there are still no general-purpose, commercial-off-the-shelf solutions integrated with secure operating systems. However, there is clearly a growing need for privacy and intellectual property protection on the Internet, as evidenced by the increasing use of FDE, recent policy directives such as the Federal Data Breach Notification Act and components of HIPAA [Brink 2009]. Between 2002 and 2007, a reported 773 breaches of U.S. organizations were reported, with a total of 267 million private records lost. More than 42% of these breaches were a result of lost or stolen hardware, including laptops, PDAs and portable memory devices [Romanosky et al. 2008]. Additionally, it is apparent that at least one major chip maker (Intel) has recognized this growing need, as two recent patent applications for adding memory encrypting hardware to processors attests [Gueron et al. 2012; Gueron et al. 2013].

The range of overheads reported in the literature is quite large (1% to 6,015%). The results on the lower end of the spectrum are possibly overly optimistic given the lack of fidelity in the simulation frameworks and the lack of standards for comparison. If standardization could be injected into the validation methodologies through common AES decryption latency, benchmarks, and so forth, it would enable more meaningful comparative analyses. Even with standardization, the number of assumptions make it difficult to be confident that simulation will provide anything more than high-level information. It ignores the more difficult and interesting implementation issues and associated security impact based on vulnerability and exploit analysis. Where, in the few cases available, the literature addresses these low-level issues, it tends to be with generalization, since there is no chance for practical experimentation or empirical

evidence [Lie et al. 2000; Shi et al. 2004; Chhabra et al. 2010]. Although the security of the encryption algorithm or cipher mode is often pointed out, it is commonly the complexity of the system in which these algorithms run that presents vulnerabilities. The most developed, although not commercially available, general-purpose technologies are FPGA soft-core emulations [Suh et al. 2007] and the Linux prototype used in Cryptkeeper [Peterson 2010]. The industrial devices are mature and practical; however, they are not general purpose, catering to highly specialized operations. Additionally, these devices are either of low frequency or expensive and difficult to upgrade [Dallas Semiconductor 1997; Arnold and Doorn 2004].

Several technologies have been incorporated into general-purpose systems recently, often without the knowledge of those buying them. These technologies include TPM chips for storing keys and encryption engines and instructions. Given a system with these components, it is now possible to experiment with memory encryption, providing an opportunity to better understand the difficult implementation details and ultimately provide data on overhead and security enhancement. This data should prove invaluable for determining the feasibility of memory encryption in general-purpose systems and for comparing against (and perhaps validating) the results of previous simulation work.

NOTICE

The U.S. Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency (DARPA) or the U.S. Government.

REFERENCES

T. Arnold, and L. Doorn 2004. The IBM PCIXCC: A new cryptographic coprocessor for the IBM eserver. *IBM Journal of Research and Development*. 120–126.

E. Barrantes, D. Ackley, S. Forrest, T. Palmer, D. Sefanovic, and D. Zovi 2003. Randomized instruction set emulation to disrupt binary code injection attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security (CCS'03)*. 281–289.

R. Best 1979. Microprocessor for executing enciphered programs. U.S. patent 4,168,396. (18 September 1979).

R. Best 1980. Preventing software piracy with crypto-microprocessors. In *Proceedings of the IEEE Spring Compcon*. 466–469.

R. Best 1981. Crypto microprocessor for executing enciphered programs. U.S. patent 4,278,837. (14 July 1981).

R. Best 1984. Crypto microprocessor that executes enciphered programs. U.S. patent 4,465,901. (14 August 1984).

A. Boileau 2006. Hit by a bus: Physical access attacks with firewire. Presented at *Ruxcon*.

D. Brink 2009. *Full-Disk Encryption on the Rise*. Aberdeen Research Group Report.

E. Casey, G. Fellows, M. Geiger, and G. Stellatos 2011. The growing impact of full disk encryption on digital forensics. *Digital Investigation* 8, 2, 129–134.

S. Chari, C. Jutla, J. Rao, and P. Rohatgi 1999. Towards sound approaches to counteract power analysis attacks. In *Proceedings of the 19th Annual International Cryptology Conference (CRYPTO'99)*. 398–412.

B. Chen, and R. Morris 2003. Certifying program execution with secure processors. In *Proceedings of the 9th Conference on Hot Topics in Operating Systems*. 23–29.

X. Chen, R. Dick, and A. Choudhary 2008. Operating system controlled processor-memory bus encryption. In *Proceedings of DATE*.

S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic 2011. SecureMe: A hardware-software approach to full system security. In *Proceedings of the International Conference on Supercomputing (ICS)*.

S. Chhabra, and Y. Solihin 2011. i-NVMM: A secure non-volatile main memory system with incremental encryption. In *Proceedings of the International Symposium on Computer Architecture (ISCA)*.

S. Chhabra, Y. Solihin, R. Lal, and M. Hoekstra 2010. An analysis of secure processor architectures. In *Transactions on Computational Science VII*. Marina L. Gavrilova and C. J. Kenneth Tan (Eds.). Lecture Notes in Computer Science. Springer-Verlag, Berlin. 101–121.

J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum 2004. Understanding data lifetime via whole system simulation. In *Proceedings of the USENIX Security Symposium*.

S. Conrad, G. Dorn, and P. Craiger 2010. Forensic analysis of a Sony Playstation 3 gaming console. In *Advances in Digital Forensics VI*. K. P. Chow and S. Shenoi (Eds.). AICT 337, 65–76.

Dallas Semiconductor. 1997. *Secure Microcontroller Data Book*. Dallas, TX.

G. Duc, and R. Keryell 2006. CryptoPage: An efficient secure architecture with memory encryption, integrity and information leakage protection. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*.

A. Dunn, O. Hofmann, B. Waters, and E. Witchel 2011. Cloaking malware with the trusted platform module. In *Proceedings of the 29th USENIX Conference on Security*. 26.

R. Elbaz, L. Torres, G. Sassatelli, P. Guillemin, C. Anguille, M. Bardouillet, C. Buatois, and J. Rigaud 2005. Hardware engines for bus encryption: A survey of existing techniques. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE)*.

W. Enck, K. Butler, T. Richardson, P. Mcdaniel, and A. Smith 2008. Defending against attacks on main memory persistence. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'08)*.

L. Gao, J. Yang, M. Chroball, Y. Zhang, S. Nguyen, and H. Lee 2006. A low cost memory remapping scheme for address bus protection. In *Proceedings of the 15th International Conference on Parallel Architecture Compilation Techniques (PACT)*.

S. Gueron 2010. *Intel Advanced Encryption Standard (AES) Instructions Set*. Intel Technical Report.

S. Gueron, G. Gerzon, I. Anati, J. Doweck, M. Maor, and L. Cho 2012. A tweakable encryption mode for memory encryption with protection against replay attacks. WO patent number 2012040679. (29 March 2012).

S. Gueron, U. Savagaonkar, F. Mckeen, C. Rozas, D. Durham, J. Doweck, O. Mulla, I. Anati, Z. Greenfield, and M. Maor 2013. Method and apparatus for memory encryption with integrity check and protection against replay attacks. WO patent number 2013002789. (3 January 2013).

P. Gutmann 2000. An open-source cryptographic coprocessor. In *Proceedings of the 2000 USENIX Security Symposium*.

J. Halderman, S. Schoen, N. Heninger, W. Clarkson, W. Paul, J. Calandrino, A. Feldman, J. Appelbaum, and E. Felten 2008. Lest we remember: Cold boot attacks on encryption keys. In *Proceedings of the USENIX Security Symposium*.

D. Hayes, and S. Qureshi 2009. Implications of Microsoft vista operating system for computer forensics investigations. In *Proceedings of the IEEE Systems, Applications and Technology Conference*. 1–9.

J. Hennessy, and D. Patterson 2006. *Computer Architecture, Fourth Edition: A Quantitative Approach*. Morgan Kaufmann Publishers, San Francisco, CA.

M. Henson, and S. Taylor 2013a. Beyond full disk encryption: Protection on security enhanced commodity processors. In *Proceedings of the 11th International Conference on Applied Cryptography and Network Security (ACNS'13)*.

M. Henson, and S. Taylor 2013b. Attack mitigation through memory encryption of security enhanced commodity processors. D. Hart (Ed.). In *Proceedings of the 8th International Conference on Information Warfare and Security (ICIW'13)*. 265–268.

D. Hong, L. Batten, S. Lim, and N. Dutt 2011. DynaPoMP: Dynamic policy-driven memory protection for SPM-based embedded systems. In *Proceedings of the Workshop on Embedded Systems Security*.

N. Howgrave-Graham, J. Dyer, and R. Gennaro 2001. Pseudo-random number generation on the IBM 4758 secure crypto coprocessor. In *Proceedings of the Workshop on Cryptographic Hardware and Embedded Systems (CHES '01)*, LNCS 2162, Springer-Verlag, 93–102.

V. Jannepally, and S. Sohoni 2009. Fast encryption and authentication for cache-to-cache transfers using GCM-AES. In *Proceedings of the International Conference on Sensors, Security, Software and Intelligent Systems*.

B. Kaplan 2007. *RAM Is Key: Extracting Disk Encryption Keys from Volatile Memory*. Master's Thesis. Carnegie Mellon University.

T. Kgil, L. Falk, and T. Mudge 2005. ChipLock: Support for secure microarchitectures. *ACM SIGARCH*, 33, 1.

P. Kocher, J. Jaffe, and B. Jun 1999. Differential power analysis. In *Proceedings of the CRYPTO 19th Annual International Cryptology Conference*. 388–397.

M. Kuhn 1988. Cipher instruction search attack on the bus-encryption security microcontroller DS5002FP. *IEEE Transactions on Computing*. 47, 1153–2257.

M. Lee, M. Ahn, and E. Kim 2007. I2SEMS: Interconnects-independent security enhances shared memory multiprocessor systems. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

D. Lie, C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz 2000. Architectural support for copy and tamper resistant software. In *Proceedings of the 9th Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 168–177.

H. Lipman, P. Rogaway, and D. Wagner 2000. Comments to NIST concerning AES modes of operations:ctr-mode encryption.

L. Martin 2010. XTS: A mode of AES for encrypting hard disks. *IEEE Security & Privacy* 8, 3 (May-June 2010), 68–69.

H. Mel, and D. Baker 2001. *Cryptography Decrypted*. Addison-Wesley, Upper Saddle River, NJ.

T. Muller, F. Freiling, and A. Dewald 2011. TRESOR runs encryption securely outside RAM. In *Proceedings of the 20th USENIX Conference on Security*.

V. Nagarajan, R. Gupta, and A. Krishnaswamy 2007. Compiler-assisted memory encryption for embedded processors. In *Proceedings of HiPPEAC*. 7–22.

D. Osvik, A. Shamir, and E. Tromer 2006. Cache attacks and countermeasures: The case of AES. In *Proceedings of the 2006 Cryptographers' Track at the RSA Conference on Topics in Cryptology*. 1–20.

P. Peterson 2010. Cryptkeeper: Improving security with encrypted RAM. In *Proceedings of the IEEE International Conference on Technologies for Homeland Security (HST)*. 120–126.

J. Platte, R. Diaz, and E. Naroska 2006. A new encryption and hashing scheme for the security architecture for microprocessors. *Communications and Multimedia Security*. 4237, 120–129.

J. Rabaiotti, and C. Hargreaves 2010. Using a software exploit to image RAM on an embedded system. *Digital Investigation*.

A. Ravi, A. Raghunathan, and S. Chakradhar 2004. Tamper resistance mechanisms for secure embedded systems. In *Proceedings of the IEEE International Conference on VLSI Design*.

B. Rogers, Y. Chenyu, S. Chhabra, M. Prvulovic, and Y. Solihin 2008. Single level integrity and confidentiality protection for distributed shared memory multiprocessors. In *Proceedings of the 14th International Symposium on High Performance Computer Architecture*. 161–172.

B. Rogers, S. Chhabra, Y. Solihin, and M. Prvulovic 2007. Using address independent seed encryption and bonsai merkle trees to make secure processors OS and performance friendly. In *Proceedings of the 40th International Symposium on Microarchitecture*. IEEE Computer Society, 183–196.

B. Rogers, M. Prvulovic, and Y. Solihin 2006. Efficient data protection for distributed shared memory multiprocessors. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT)*.

B. Rogers, Y. Solihin, and M. Prvulovic 2005. Memory predecryption: Hiding the latency overhead of memory encryption. *ACM SIGARCH Computer Architecture News*, 33, 1 (March 2005), 27–33.

S. Romanosky, R. Telang, and A. Acquisti 2008. *Do Data Breach Disclosure Laws Reduce Identity Theft?* Carnegie Mellon Technical Report.

W. Shi, H. Lee, M. Ghosh, and C. Lu 2004. Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems. In *Proceedings of the 13th International Conference on Parallel Architecture and Compilation Techniques (PACT)*.

P. Simmons 2011. Security through amnesia: A software-based solution to the cold boot attack on disk encryption. In *Proceedings of the 27th Annual Computer Security Applications Conference*.

S. Smith 2004. Magic boxes and boots: Security in hardware. *IEEE Computer Software* 37, 10, 106–109.

M. Steil 2005. 17 mistakes Microsoft made in the Xbox security system. In *Proceedings of the 22nd Chaos Communication Congress*.

M. Steil, and F. Domke 2008. The Xbox 360 Security System and Its Weaknesses. Google TechTalk, Available at http://www.youtube.com/watch?v=uxjpmc8ZIxM.

L. Su, S. Courcambick, P. Guillemin, C. Schwarz, and R. Pascalet 2009a. SecBus: Operating system controlled hierarchical page-based memory bus protection. *EDAA*.

L. Su, A. Martinez, P. Guillemin, S. Cerdan, R. Pacalet 2009b. Hardware mechanism and performance evaluation of hierarchical page-based memory bus protection. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE)*.

G. Suh, D. Clarke, B. Gassend, M. Dijk, and S. Devadas 2003. Aegis: Architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th International Conference on Supercomputing*.

G. Suh, D. Clarke, B. Gassend, M. Dijk, and S. Devadas 2005. Efficient memory integrity verification and encryption for secure processors. In *Proceedings of the 36th International Symposium on Microarchitecture*.

G. Suh, C. O'Donell, and S. Devadas 2007. Aegis: A single-chip secure processor. In *IEEE Design and Test of Computers*.

G. Vandana 2008. *Exploring Trusted Platform Module Capabilities: A Theoretical Experimental Study*. Ph.D. Dissertation.

C. Yan, B. Rogers, D. Englender, Y. Solihin, and M. Prvulovic 2006. Improving cost performance and security of memory encryption and authentication. In *Proceedings of the 33rd International Symposium on Computer Architecture*.

J. Yang, L. Gao, and Y. Zhang 2005. Improving memory encryption performance in secure processors. *IEEE Transactions on Computing*.

Y. Zhang, L. Gao, J. Yang, X. Zhang, and R. Gupta 2005. SENSS: Security enhancement to symmetric shared memory multiprocessors. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*.

X. Zhuang, T. Zhang, and S. Pande 2004. Hide: An infrastructure for efficiently protecting information leakage on the address bus. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 72–84.

Received April 2013; revised September 2013; accepted October 2013