SDSM: Fast and Scalable Security Support for Directory-Based Distributed Shared Memory

Ofir Shwartz and Yitzhak Birk
Electrical Engineering Department
Technion - Israel Institute of Technology, Haifa, Israel
{ofirshw@tx, birk@ee}.technion.ac.il

***Abstract*— Secure computation is increasingly required, most notably when using public clouds. Many secure CPU architectures have been proposed, mostly focusing on single-threaded applications running on a single node. However, security for parallel and distributed computation is also needed, requiring the sharing of secret data among mutually trusting threads running in different compute nodes in an untrusted environment. We propose SDSM, a novel hardware approach for providing secure directory-based distributed shared memory. Unlike previously proposed schemes that cannot maintain reasonable performance beyond 32 cores, our approach allows secure parallel applications to scale efficiently to thousands of cores.**

Keywords—security; memory; coherence; multi-core; DSM; directory based coherence; shared memory

I INTRODUCTION

A. The Problem

Security and privacy in computer systems are major concerns, especially when running programs on third-party systems such as public clouds, in which the user may be unwilling to trust the system provider and thus its hypervisor, operating system and even most of its hardware.

Various studies [1,2,5,6,7] and industry efforts (E.g., SecureBlue++ [4], Intel SGX [3]) have attempted to maintain the secrecy of a program running on a platform of an untrusted owner. They all share a basic element: using encryption to preserve program secrecy.

To execute an encrypted program, every secure system has a system-internal *trusted area* (TA) holding the data (and similarly, code) as cleartext, because operations that can be performed directly on encrypted data are currently very limited [9]. The TA commonly includes the processing core as well as data and instruction caches, and related control mechanisms. Information is encrypted by the user before being provided to the machine in the first place; it is decrypted upon entering the TA (into the cache), is encrypted automatically upon eviction, and is decrypted automatically whenever fetched back into the cache. Unfortunately, all this comes at the cost of added latency.

For encryption of data being placed in (untrusted) memory, *counter mode encryption* [11] has been proposed. Data is encrypted using a keystream block (KB) [10]. The KB is calculated as a complex cryptographic function of the block address, a secret key, and a counter seed. Upon cache miss, the requested block's address is known; assuming the presence of the key and the seed in the TA, the decryption KB is calculated while the missing block is being fetched, requiring only a bitwise XOR with the arriving encrypted block. The memory access latency is thus used to hide that of the KB preparation. However, data encryption upon cache eviction has remained slow, as the KB calculation traditionally [11,12] requires the address of the block being evicted, which is not known in advance. Alternatively, one can pre-calculate and store in the TA dedicated KBs for some of the cache lines (100% memory overhead for those).

Most past studies assigned little importance to encryption latency of evicted blocks, claiming (rightfully) that a write buffer within the TA can be used to hide this latency. However, in a many-core system, eviction may take place in response to a request for the cache block by another compute core. If the cores are connected through an untrusted medium (E.g., PCB), the data must be encrypted for eviction, adding noticeable latency to block fetching. We therefore focus on this problem.

General setting. We consider a multi-node distributed shared memory (DSM) system with a (per process) shared address space. Each node consists of a core with its private cache and memory, and the memory coherence of the system is managed by a central or distributed directory. The directory may be implemented in hardware or in software, but it must be trusted (discussed later). At any given time, a given block may reside in multiple private memories and/or caches with read-only permission. Once write permission is granted, a block may only reside in one private memory and its local cache. When a block is needed by a core that does not have it in its own memory, the core's hardware turns to the directory for assistance.

Due to unacceptable communication latency, many DSM systems scale to thousands of computing nodes by sacrificing memory coherence. However, recent technologies (such as Compass-EOS' icPhotonics [23]) allow low-latency communication that may enable coherent DSM systems with thousands of cores at the box or rack level, simplifying the programming model for many massively parallel applications. Fast security support for such systems is thus of interest.

Throughout this paper, we use *core* to denote a single threaded execution unit along with its private caches, security access control, and cryptography primitives. (Single-threaded merely for facility of exposition.) We refer to the core requiring a missing data block as the *requestor*, and to the core holding this block as the *sender*. Although each core serves as both *sender* and *requestor*, we discuss these roles separately.

This work was supported in part by the Hasso Plattner Institute.

978-1-4673-8826-9/16/$31.00 ©2016 IEEE

B. Threat Model

We consider a DSM system for many cores connected through an untrusted medium. A parallel program (that shares data among its threads) runs on the system, requiring a common key to be stored in the executing cores. A setup process is assumed to exist for securely distributing and storing such keys in the cores. The cores are trusted, and we assume that they are correct and their internals are physically inaccessible for snooping. An attacker with physical access to the system can inspect and record any off-chip signal and message. It can also change messages, replay old ones and initiate new ones. This applies to both data, commands, control, and coherence management messages (inter-core or between cores and main memory).

All other software, including other concurrently running applications, operating system and hypervisor are assumed to be hostile. We rely on a secure CPU architecture (such as Intel SGX) enforcing by hardware correct process separation, permissions and data integrity using compartmentalized state within the TA; data alteration by hardware, as well as software security issues are treated by other layers, as part of a secure architecture (such as SGX). Furthermore, we do not create new problems in that respect. Denial of service of any kind is outside the scope, as an attacker with physical access may simply power the system down. Side channel attacks are also outside the scope, but we do not introduce new vulnerabilities. These settings are common in real world scenarios, and similar settings were addressed in [15,16,19].

In this setting, we strive to provide fast, scalable security support for directory-based coherent distributed shared memory. Security includes preserving the secrecy of the user's program and data, and detecting any alterations thereof.

C. Our Contributions

We present a new approach for supporting secure coherent distributed shared memory (SDSM), which provides support for secrecy and integrity of inter-core communication. SDSM scales to thousands of cores while maintaining good performance. It can be added to secure CPUs using any variant of counter mode encryption, running either a trusted or an untrusted OS, such as SGX [3]. By using a TCM (a trusted coherence manager, comprising a trusted directory with added functionality), exploiting native latencies of the DSM system, and using a simple adaptation technique, we are able to dramatically reduce wasted work relative to prior art; also, SDSM scales with essentially constant per-core hardware resources. Throughout this paper, we assume a write-back cache and inclusive main memory, updating only modified blocks. We do not focus on any particular coherence mechanism, but consider MESI [28] as a common yet simple example.

The specific contributions of this paper are:

- A new approach for using seeds in counter mode encryption with block-address independent KBs, obviating the need to supply initial seeds while preventing initial KBs from being reused during runtime.
- A new seed management and distribution protocol for avoiding wasted KB pre-calculation work, and exploiting DSM systems' communication latency for hiding that of the KB calculations.
- Smart allocation of hardware resources to obtain a secure and scalable DSM with essentially constant per-core resources.
- Establishing the need for a trusted coherence manager (TCM) to ensure correct coherence status and messages.

The remainder of the paper is organized as follows. Section II provides an overview of memory encryption and related work; Section III presents our contributions; Section IV evaluates them, and section V offers concluding remarks.

II Backgroudn and Related Work

This section reviews related work on counter mode memory encryption, mechanisms for memory encryption in multi-core settings, encryption seed management and encryption latency reduction schemes.

A Memory Encryption

Many systems [15,16,19] use Galois Counter Mode (GCM) [24], which is an authenticated variant of counter mode encryption. GCM relies on a running counter, with KB generation requiring a long computation, similar to counter mode encryption. For simplicity, we will consider the original counter mode as our encryption algorithm, but the ideas and results are easily adaptable to any of its variants.

The use of keystream blocks for memory encryption simply entails encrypting k-bit data D using a k-bit pseudo-random secret R by performing a bitwise XOR: $E = D \oplus R$. XOR is reversible: $D = E \oplus R$, and is fast to execute.

Recent implementations of counter mode encryption [25] use AES [20] block cipher to generate a pseudo-random number $R = Enc(P, k)$, where P is a block-related seed, and k is a symmetric secret key. P is commonly defined as $P = VA||S$ [11], which is the concatenation of the block's virtual address (VA) with S, a counter based seed. Having a unique VA ensures that each block has a unique set of P values, so AES guarantees that using the same key k, R is unique per block and does not repeat as long as S doesn't. The seeds may be stored in the clear, as no attacker can reproduce R without knowing the secret key k.

Only S must be stored per evicted block, along with negligible-sized metadata for locating it based on the block address. Together, their size is only a small fraction of R's, resulting in reasonable storage overhead. (See [1,8,26,21] for seed storage and caching details.)

Enc may be any block cipher algorithm, and P is padded with zeros up to the required size of *Enc*'s input. If *Enc*'s output is shorter than the data block, we use multiple *Enc* blocks $R_i = Enc(P_i, k)$, where $P_i = P \,||\, block\_index$, and concatenate all R_i's to form a KB of the required size. [11]. The encryption's strength is the same as the block cipher's. The seeds are commonly initialized to 0, obviating the need for supplying initial values while maintaining a unique KB for each block.

Using a proper design, the VA and seed of a missing block are known in the TA at the time of a fetch request. (For simplic-

ity, all the caches are assumed to be in the TA.) The calculation of the *Enc* function, typically shorter than the main memory latency, is initiated concurrently with the data fetch request, so the latency calculating the KB required for decryption of the fetched block is hidden by the memory latency.

B Multi-Core: KB Pre-Generation and Encryption Latency Reduction

In multi-core settings, fast memory encryption is essential. Without it, remotely requested blocks will suffer from high eviction latency, which adds to their fetch latency. This problem was first addressed in [13] and subsequently in [14], but only for bus-based shared memory multiprocessor architectures, which are very restrictive and non-scalable.

In order to address distributed directory-based shared memory settings, [19] suggested that each sender core pre-generate a seed and use it to pre-calculate a block-address independent KB. Upon eviction, this KB may be used to encrypt the block that is being evicted on the fly.

A later work [15] added a central trusted global counting controller (GCC) for all the processing cores, providing a trusted running counter for generating the seeds. It also added three buffers in each core: outstanding pre-calculated KBs (sender), KBs of recently fetched blocks (for re-encryption upon eviction if unchanged), and outstanding KBs for incoming blocks (requestor); this provides greater flexibility with stressful workloads. Its threat model assumes tamper-free memory management messages, allowing only data messages to be attacked, and that no management or coherence message is ever lost.

In order to hide the decryption latency, a requestor must receive the seed that served to produce its encryption KB so as to permit calculation of the KB before the encrypted data arrives. [15,19] suggested sending the outstanding seeds to all cores in the DSM, so they can prepare the KBs in advance. As any KB may only be used to encrypt a single data block, in an N-core system each useful block transfer is accompanied by N-2 wasted seed transfers and KB calculations; this unacceptable waste of energy, memory and bandwidth moreover grows with the number of cores. Limiting the cache size for seeds and KBs at the requestors is also problematic, as it would reduce hit rate as cores are added.

In [16], a DSM scheme with no KB pre-generation is discussed. Each block modification triggers the creation of an outstanding KB, kept temporarily with its new seed. Due to the large area needed per KB entry (roughly the size of a cache block), only a limited amount of buffer space is used for these KBs; if not found in the buffer, a lengthy process is required for fetching the current seed from memory and calculating a new KB. (KBs must not be kept outside the TA!)

Lastly, [16] protects the integrity of seed and control messages by using a delayed timestamped message authenticating code (MAC), calculated using a cryptographic keyed hash function [27] with the program's private key. It is sent back as ACK to these messages (piggybacked onto the next message) without adding latency to the critical path, so these messages either arrive correctly or a tamper event is declared. Data integrity is protected using GHASH [24], which is a lightweight MAC of GCM. We adopt the same integrity mechanisms in our work.

All previous works failed to provide schemes that scale to many-core settings. Their total hardware resource requirement grows quadratically with the number of cores (because each core holds a set of every other core's KBs), or else the efficiency of the existing resources drops dramatically as the core count increases. Previous works did not discuss the trust model for the coherence manager (such as a directory) for producing correct query replies. Finally, they did not address multiple concurrently executing secure applications; there, the limited resources cannot be replicated per application, so intelligent management is a must. We address all these issues.

C Seed Management

In [15,19], block-address-independent KBs are used; this allows them to prepare one KB in advance and use it to encrypt the next evicted block. As the secret key is the same for all memory blocks, preventing KB reuse requires the use of different seeds for different blocks. To this end, a global counter (GCC) was suggested, such that each seed value is only used once at runtime [15].

For initial delivery to the secure machine, the data and instructions of a secret program are encrypted using initial KBs. Concatenating the block-address with the seed to form the KB (see III.A.) allows the use of *zero* as the initial seed value, obviating the need for providing the seed with the program. However, using block-address independent KBs raises a new issue: supplying initial unique-per-block seeds with the program (while forcing the GCC to refrain from using these values) will cost additional storage. [15,19] did not address the seed repetition problem presented here, nor did they discuss how initial seeds are supplied. We will present a new method that uses *no* initial seeds while still distinguishing among KBs of different blocks and avoiding KB reuse at any time.

III SDSM

In this section we present SDSM, our scheme for providing fast and scalable security support for directory-based distributed shared memory. We first present its methods and building blocks, and then show how these are put together.

A Seeds and Keystream Blocks

Our goal is to use block-address independent KBs, while using unique KBs both at the beginning and during runtime. We present a simple yet novel approach for choosing seed values, deriving KBs form these seeds, and a heuristic for when to do so. Our scheme hinges on the observation that block-address independence is only required for modified blocks during runtime (for KB pre-generation), not for the initial encryption. This may be used in any secure system or CPU in conjunction with any variant of counter-mode encryption.

We use $R = ENC(P,k)$ as the KB for encrypting the blocks, with $P = f(S,VA)$ padded by zeros to match the required input size of ENC. The function f is defined as follows:

$$f(S,VA) = \begin{cases} 0\ldots00\ ||VA & if\ S = 0 \setminus\setminus initial\ seed\ value \\ 0\ldots01\ ||S & otherwise \setminus\setminus VA - independent, \end{cases}$$

with the seeds S at least the size of VA. Each block is initially

encrypted using its VA as the P parameter of $ENC(P,k)$; in subsequent encryptions, P is VA-independent and consists of the value of the seed concatenated by '1', so the resulting KBs differ from the initial KBs. Upon decryption, checking if the seed is equal to zero (and acting accordingly) takes negligible time. The seeds are initialized to zero. During runtime, the seeds are simply generated by a running counter, which increments upon each eviction of a modified block. (In Section III.D we discuss assigning and using these seeds with many secure CPUs, where we also increment the write counter for non-modified blocks.)

B Process-aware Keystream Block Pre-generation

DSM systems usually serve multiple applications concurrently; yet, past work didn't consider concurrent secure processes. In SDSM, each sender core receives seeds from the write counter or directory (discussed in III.D), but pre-generates its own outstanding KBs using these seeds. Requestor cores generate KBs based on senders' seeds (received from the directory) and on the keys shared with them.

Each sender uses a dedicated cache for holding outstanding pre-calculated KBs (as in [15]), which are subsequently used for encrypting remotely-requested blocks. Each secure process uses its own secret key (shared by its threads), so the sender should prepare outstanding KBs **per process**. Each sender core monitors past block requests, and learns from which of its secure processes they were requested recently. Then, the sender prepares outstanding KBs for these processes, favoring those that were asked for more blocks recently.

Assuming constant per-core resources (regardless of the number of cores), and that each core may run many secure programs concurrently, this approach helps in better utilizing the core's resources (compared both with generating KBs for every process running in the system and with so doing for every process that currently has active blocks in its cache).

C Trusted Coherence Manager (TCM)

Any distributed coherent memory system has a managing entity (such as a directory) that keeps the status of memory blocks and responds to queries about it. Some previous works ([15,19]) assumed that the related management messages are delivered correctly; [16] suggested a method for ensuring message integrity of both counter and coherence messages. However, to our knowledge, an adversarial coherence manager (directory) was not considered before. We next do so.

Proposition: a trusted coherence manager is mandatory.

Proof: Consider cores A, B, and C. Cores A and B share a block for read. Core A wishes to modify this block, but the adversarial coherence manager refrains from sending an invalidation message to B. When C requests this block, it may get an old version from B. [16]'s inter-core message integrity is thus insufficient with an adversarial coherence manager. □

Any coherence manager may be used; it simply needs to be placed within a trusted area. A distributed network of TCMs may be used, wherein each manages an address-based fraction of the memory, as long as all these are trusted and we ensure that messages are verified for integrity and their reception is acknowledged. We also use the TCM for managing a per-process universal (write) counter for supplying unique seeds, and in the next subsection we will discuss using the TCM as part of the seed management system. With a distributed network of TCMs, for each secure process the seed-generation range must be partitioned among the TCMs to avoid duplication. Each TCM is thus assigned a unique portion of the seed-value space and, for every process, a unique portion of the virtual address space.

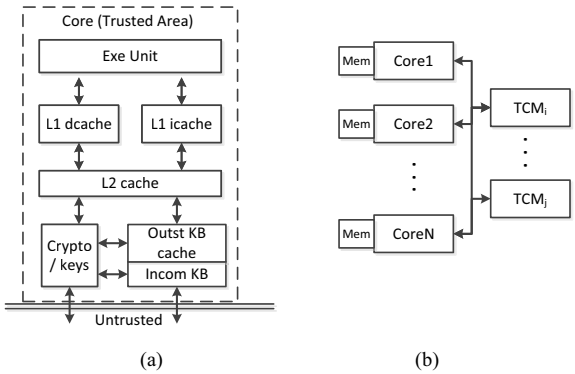

Fig. 1.(a) Core architecture; (b) System architecture.

D Putting it all together

We now present our hardware requirements and scheme for fast and scalable secure data sharing for a coherent DSM system, incorporating the aforementioned building blocks (some for correctness and others for performance and efficiency). Specifically, we present a scheme whereby 1) KB calculation latency is hidden from the requestor; 2) little work is done for KB calculation; and 3) a small amount of hardware resources suffices even for large systems.

We consider a DSM system built of many secure CPU cores (referred to in the threat model). Each core includes a trusted area used for the following tasks: it securely stores the application's secret keys (Section I.A.); it implements GCM encryption, and treats seeds as described in III.A. KB calculation time is assumed to be less than the core's **local** memory access time (though not required for correctness). It has a small cache for outstanding KBs (for sending), allocated per process as described in III.B, and a **single** KB entry for receiving blocks (unlike [15]'s KB cache). It implements [16]'s integrity mechanisms (described in Section II.B.), taking its latency out of the critical path. Fig. 1.a depicts our core architecture. One or more TCMs may serve the cores, and each TCM is responsible for a subset of the memory address space. (Fig. 1.b).

Seed Management in SDSM

Each cache miss results in a request sent to the requested block's home TCM for checking the block's status. This request also states its purpose – read or write. The TCM forwards the request to one of the cores presently possessing a current copy of the block. (When in shared read mode, the TCM may have a choice.) Considering the communication latency of requests between different cores and the time to calculate the KB, the KB calculation latency can be hidden from the requestor if 1) it gets the seed before it gets the encrypted data such that its KB

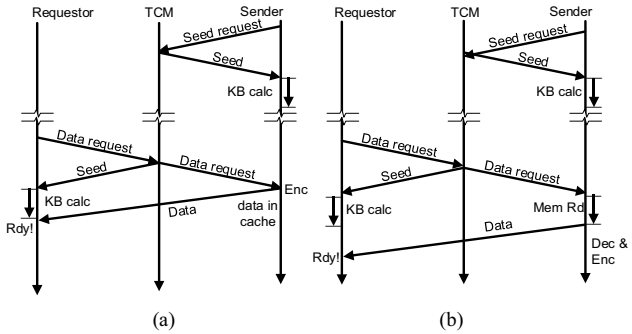

Fig. 2. Data read procedure: (a) sender cache hit; (b) sender cache miss.

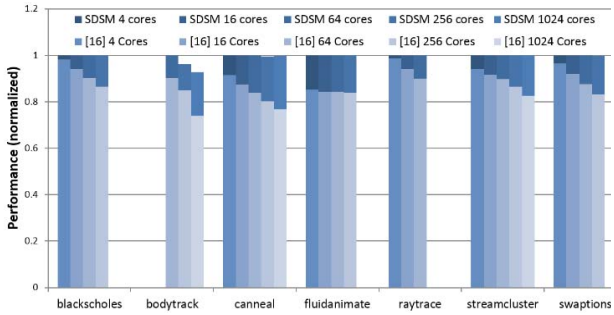

Fig 3. (a) SDSM performance relative to [16], normalized to no security

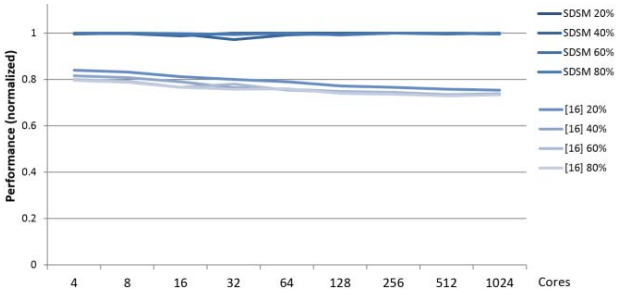

Fig 3. (b) Normalized performance comparison for synthetic benchmarks with various miss rates.

calculation won't delay its block decryption, and 2) the block owner uses pre-calculated KBs to instantly encrypt blocks that are being evicted. We do this as follows:

(1) The TCM generates unique seed(s) (using a per-process counter) at a sender's request (for seeds), stores them in its (TCM's) cache, and sends a copy to the sender. Each sender maintains a short list of (block-address independent) seeds, and prepares outstanding KBs for future evictions.

(2) A modified block that is evicted from its owner's cache is encrypted using an outstanding KB (calculated using a seed previously received from the TCM).

(3) Once the TCM receives a block request, it sends the first in line (oldest) outstanding seed of the block's owner directly to the requestor (latency reduction), and forwards the request to the block owner, suggesting the sender's seed to be used. (This addresses race conditions among simultaneous requests). If the block is in the sender's cache, it promptly encrypts it using the pre-generated KB, and sends it to the requestor (Fig 2.a). If it isn't, the sender loads it from its memory, decrypts (using its cached seed with no added latency), re-encrypts with the pre-generated KB, and sends to the requestor (Fig 2.b). The sender may avoid re-encryption and simply send the currently used seed to the requestor (with the encrypted block); however, the requestor will not be able to use a pre-generated KB, resulting in additional undesirable latency.

Because we use the inherent communication latency to hide the seed transfer latency and KB generation latency at the requestor, the requestor only needs to prepare a useful KB upon need at no latency cost.

Unlike previous work, we do not use a per-core dedicated KB cache for decrypting incoming blocks; instead, the TCM holds the seeds in its cache. Consumer CPUs presently have up to 8MB caches, and server CPUs may have ten times more [22]. Using 8-byte seeds and only half of the TCM cache for seeds (the rest is used for coherence management), we have roughly 0.5M seeds available in the TCM cache. Assuming 10 outstanding seeds per sender, as far as cache goes, one TCM can serve up to 50k cores while providing seeds from cache. The compute requirement of a TCM is similar to that of a conventional directory.

The number of cores per TCM may vary to match the expected workload and required TCM hit rate. Considering an extremely high hit rate for seeds in the TCM cache, our approach shows similar performance to that of a directory based system with no security at all.

IV Evaluation

In this section we evaluate SDSM for performance and scalability, and compare it with state-of-the-art work. We ran the PARSEC benchmark suite [18], focusing on applications that can scale to hundreds of threads. We used Pin [17] to capture the benchmarks' activity, and added our seed management and communication layer. Each benchmark was executed with no security layer as the baseline for performance measurements, with [16]'s scheme (which to our knowledge has the most recent results published for similar settings), and with SDSM. The performance results are normalized to the baseline. In our tests we used 100 clock cycles for core-to-core latency, and 80 clock cycles for calculating the keystream block [16]. We used 10 outstanding KBs per sender. Unlike [16], which assumed higher communication latency as the core count grew, our evaluations of all the schemes assumed the same latency for every setting, so the security related overheads are fully exposed.

We clearly see (Fig. 3.a) that SDSM scales easily to a thousand cores with less than 2.5% performance reduction, and less than 0.8% performance reduction with 256 cores. [16]'s performance drops by 22% with 1,000 cores, and by 16% with 256 cores. We saw no performance improvement when increasing the number of outstanding KBs per sender beyond 10. In [16], a sender only caches block-specific KBs for a few recently modified blocks, assuming that these are going to be requested soon. This approach suffers from a KB miss rate that

increases as the number of potential requestors increases. We only assign a KB for a requested block upon request, so we practically never miss any KB.

Next, we used synthetic benchmarks to assess the system's behavior with various miss rates and core counts. Fig. 3.b shows that SDSM exhibits less than 0.3% performance degradation for any core count and miss rate, whereas [16] is sensitive to both (up to 25% performance reduction).

We repeated with three different communication latencies: 50, 100, and 200 clock cycles. As expected, the performance penalty for calculating the keystream blocks (by the prior art schemes) drops as the communication latency rises (Amdahl's law). We also evaluated the extra traffic caused by our scheme and found it to be similar to [16].

Overheads. Storage overhead is smaller than previously suggested architectures for the same settings [15,16,19]. The main reason is that we only keep a small cache for outstanding KBs (holding 10 entries) that will be used effectively, and only a single incoming KB per core. The number of cores assigned for TCMs is the same as would be used for a conventional directory, so we do not create new overheads there.

V Conclusions

We presented SDSM, a novel approach for creating a scalable, secure distributed directory-based shared memory system. Exploiting native latencies of the DSM system, we are able to scale to thousands of cores, with a tiny performance degradation relative to non-secure DSMs. We are also able to avoid redundant work (relative to previous work), and thus save energy and make better use of memory space. SDSM will enable the construction of massively parallel secure and efficient directory-based coherent memory systems.

Future work includes more detailed study of per-core resource requirements (e.g., energy and traffic) for an N-core parallel task, optimizations for non-uniform memory access (NUMA) systems, and supporting dynamically changing systems.

References

[1] D. L. C. Thekkath, M. Mitchell, P. Lincoln, D. Boneh, J. Mitchell, and M. Horowitz, "Architectural support for copy and tamper resistant software," ACM SIGOPS Oper. Syst. Rev., pp. 168–177, 2000.

[2] G. E. Suh, D. Clarke, B. Gassend, M. Van Dijk, and S. Devadas, "AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing," Proc. Int. Conf. Supercomput., pp. 160–171, 2003.

[3] I. Anati, S. Gueron, S. Johnson, and V. Scarlata, "Innovative Technology for CPU Based Attestation and Sealing," HASP - Proc. 2nd Int. Work. Hardw. Archit. Support Secur. Priv., pp. 1–7, 2013.

[4] P. Williams and R. Boivie, "CPU Support for Secure Executables," in Trust and Trustworthy Computing, vol. 6740, pp. 172–187, 2011.

[5] D. Evtyushkin, J. Elwell, M. Ozsoy, D. Ponomarev, N. A. Ghazaleh, and R. Riley, "Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution," 2014 47th Annu. Int. Symp. Microarchitecture, MICRO, pp. 190–202, 2014.

[6] S. Chhabra, B. Rogers, Y. Solihin, and M. Prvulovic, "SecureME : A Hardware-Software Approach to Full System Security," Proc. Int. Conf. Supercomput., pp. 108–119, 2011.

[7] X. Chen, T. Garfinkel, E. C. Lewis, P. Subrahmanyam, C. a Waldspurger, D. Boneh, J. Dwoskin, and D. R. K. Ports, "Overshadow: a virtualization-based approach to retrofitting protection in commodity operating systems," Proc. 13th Int. Conf. Archit. Support Program. Lang. Oper. Syst. - ASPLOS XIII, p. 2, 2008.

[8] B. Rogers, S. Chhabra, Y. Solihin, and M. Prvulovic, "Using address independent seed encryption and bonsai merkle trees to make secure processors OS-and performance-friendly," in Proceedings of the Annu. Int. Symp. on Microarchitecture, MICRO, pp. 183–194, 2007.

[9] C. Gentry, "Fully homomorphic encryption using ideal lattices," Proc. 41st Annu. ACM Symp. Symp. theory Comput. STOC, p. 169, 2009.

[10] D. a McGrew, "Counter Mode Security : Analysis and Recommendations," pp. 1–8, 2002.

[11] J. Yang, Y. Zhang, and L. Gao, "Fast secure processor for inhibiting software piracy and tampering," in Proceedings of the Annu. Int. Symp. on Microarchitecture, MICRO, pp. 351–360, 2003.

[12] G. E. Suh, D. Clarke, B. Gasend, M. Van Dijk, and S. Devadas, "Efficient memory integrity verification and encryption for secure processors," in Proceedings of the Annu. Int. Symp. on Microarchitecture, MICRO, pp. 339–350, 2003.

[13] W. Shi, H.-H. S. Lee, M. Ghosh, and C. Lu, "Architectural support for high speed protection of memory integrity and confidentiality in multiprocessor systems," Proceedings. 13th Int. Conf. Parallel Archit. Compil. Tech, PACT, pp. 123 – 34, 2004.

[14] Y. Zhang, L. Gao, J. Yang, X. Zhang, and R. Gupta, "SENSS: Security Enhancement to Symmetric Shared memory multiprocessors," in Proceedings - International Symposium on High-Performance Computer Architecture, pp. 352–362, 2005.

[15] M. Lee, M. Ahn, and E. J. Kim, "I2SEMS: Interconnects-independent security enhanced shared memory multiprocessor systems," Proc. 13th Int. Conf. Parallel Archit. Compil. Tech, PACT 16th, pp. 94–103, 2007.

[16] B. Rogers, C. Yan, S. Chhabra, M. Prvulovic, and Y. Solihin, "Single-level integrity and confidentiality protection for distributed shared memory multiprocessors," in Proceedings - International Symposium on High-Performance Computer Architecture, pp. 161–172, 2008.

[17] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: building customized program analysis tools with dynamic instrumentation," Proc. 2005 ACM SIGPLAN Conf. Program. Lang. Des. Implement. - PLDI, p. 190, 2005.

[18] C. Bienia and K. Li, "Parsec 2.0: A new benchmark suite for chip-multiprocessors," 5th Annu. Work. Model. Benchmarking Simul., pp. 1–9, 2009.

[19] B. Rogers, M. Prvulovic, and Y. Solihin, "Efficient data protection for distributed shared memory multiprocessors," Proc. 13th Int. Conf. Parallel Archit. Compil. Tech, PACT 15th, pp. 84–94, 2006.

[20] N. I. of Science and Technology. FIPS PUB 197: Advanced Encryption Standard (AES), November 2001.

[21] C. Yan, B. Rogers, D. Englender, Y. Solihin, and M. Prvulovic, "Improving cost, performance, and security of memory encryption and authentication," in Proceedings - International Symposium on Computer Architecture, pp. 179–190, 2006.

[22] J.Turley, "White Paper Introduction to Intel® Architecture" http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/ia-introduction-basics-paper.pdf

[23] S. Benjamin, "Chip to chip optical interconnect assembly", http://www.semiconwest.org/sites/semiconwest.org/files/data14/docs/SW2014_%20Shuki%20Benjamin_Compass%20EOS.pdf

[24] D. A. Mcgrew and J. Viega, "The Galois / Counter Mode of Operation (GCM) Intellectual Property Statement," vol. 67, no. 3, p. 265279, 2004.

[25] M. Hanifdurad, M. N. Khan, and Z. Ahmad, "Analysis and Optimization of Galois / Counter Mode (GCM) using MPI," no. Gf 2128, pp. 333–337, 2015.

[26] B. Gassend, B. Gassend, G. E. Suh, G. E. Suh, D. Clarke, D. Clarke, M. van Dijk, M. van Dijk, S. Devadas, and S. Devadas, "Caches and Hash Trees for Efficient Memory Integrity Verification," Proc. Ninth Int. Symp. High-Performance Comput. Archit., pp. 295–306, 2003..

[27] H. Krawczyk, M. Bellare, and R. Canetti, "RFC2104 - HMAC: Keyed-hashing for message authentication," 1997.

[28] M. S. Papamarcos and J. H. Patel, "A low-overhead coherence solution for multiprocessors with private cache memories," ACM SIGARCH Comput. Archit. News, vol. 12, pp. 348–354, 1984.