Implementing Encryption as an Architectural Approach to Securing Memory Systems

Ina Papadhopulli, Amina Sokoli
Department of Computer Engineering, Polytechnic University of Tirana, Albania

Abstract— **Ensuring secure memory systems is critical in mitigating hardware-level threats in modern computing. This work investigates the integration of encryption into memory architectures as a defensive mechanism against physical attacks such as RowHammer and Cold Boot. A custom encryption engine is implemented within Ramulator 2.0, an open-source DRAM simulator, to enable evaluation of performance-security trade-offs. The impact of encryption on memory latency is quantified, and system-level overheads are analyzed. Additionally, optimization strategies involving memory scheduling are explored to reduce timing penalties associated with encryption. The results demonstrate that encryption can enhance memory protection while introducing architectural challenges related to latency and integration.**

Keywords— **Hardware Security, Memory Protection, data integrity, AES, Ramulator 2.0, Performance Overhead, Memory Scheduling**

I. INTRODUCTION

Secure memory systems are fundamental to the integrity and confidentiality of modern computing architectures, yet they remain vulnerable to hardware-based attacks such as RowHammer and Cold Boot exploits. Traditional memory protection mechanisms, including Error-Correcting Codes (ECC) and Address Space Layout Randomization (ASLR), provide limited defense against these sophisticated threats, often failing to prevent unauthorized data access or manipulation. On the other hand, cryptographic techniques, such as memory encryption, offer robust protection by rendering data unreadable without the appropriate decryption key, but their implementation can introduce significant performance overhead, particularly in latency-sensitive applications. To address these challenges, this paper proposes an encryption-based architectural strategy that integrates the Advanced Encryption Standard (AES) algorithm with an optimized memory scheduler to enhance security while minimizing performance degradation.

Building upon prior work on memory system security, this study extends the exploration of encryption as a protective measure by implementing an AES-based encryption plugin within the Ramulator 2.0 open-source memory simulator. Unlike previous approaches that focus on hardware-level encryption, such as Intel Total Memory Encryption (TME) or AMD Secure Memory Encryption (SME), this work introduces a software-driven model that operates at the memory controller level, enabling real-time encryption and decryption of data during read and write operations. The proposed model explicitly addresses the trade-offs between security and performance, particularly in the context of DRAM-based systems.

The proposed framework consists of two key components:

- An AES encryption plugin integrated into the Ramulator 2.0 simulator, which employs a 128-bit key to encrypt and decrypt memory transactions, ensuring protection against hardware attacks
- An optimized memory scheduler, termed `EncryptionScheduler`, designed to reduce the latency overhead introduced by cryptographic operations by efficiently managing memory access requests.

The model targets primary memory variables, including data integrity and access latency, which are critical indicators of system security and performance. By simulating memory operations under various

workloads, the framework evaluates the effectiveness of encryption in mitigating hardware threats while maintaining acceptable performance levels.

The evaluation includes comparisons with baseline configurations with and without encryption, providing a comprehensive analysis of the proposed approach's benefits and limitations.
The structure of the paper is as follows. Section 2 provides an overview of related work on memory system vulnerabilities, cryptographic techniques, and memory simulators. Section 3 details the proposed AES encryption plugin and `EncryptionScheduler`, along with a summary of the implementation framework and the comparison of the results based on the configurations with and without the optimized scheduler. Finally, Section 4 concludes with key findings and recommendations for future research in secure memory architectures.

II. RELATED WORK

The security of memory systems has been a focal point of research due to the increasing prevalence of hardware-based attacks such as RowHammer and Cold Boot exploits. These vulnerabilities exploit the physical properties of Dynamic Random Access Memory (DRAM) to manipulate or extract sensitive data, necessitating robust countermeasures. Existing literature can be broadly categorized into hardware-based and software-based approaches to memory protection, with a growing emphasis on cryptographic techniques and simulation-based analysis to evaluate their efficacy.

Early work on memory security focused on mitigating transient errors in DRAM. Patterson and Hennessy [1, 2] outlined the hierarchical structure of memory systems, highlighting their susceptibility to soft errors caused by environmental factors like alpha particle emissions. Techniques such as Error-Correcting Codes (ECC) and memory scrubbing [21] were developed to detect and correct bit-flips, but these methods are insufficient against deliberate attacks like RowHammer, which induces bit-flips through repeated memory accesses [5]. RowHammer exploits can bypass traditional protections, prompting the exploration of more advanced countermeasures.
Hardware-based encryption solutions have emerged as a prominent strategy for securing memory systems. Intel's Total Memory Encryption (TME) and Multi-Key TME (TME-MK) utilize AES-XTS to encrypt the entire RAM or specific memory pages, respectively, with minimal performance overhead [16, 19, 20]. Similarly, AMD's Secure Memory Encryption (SME) and Secure Encrypted Virtualization (SEV) provide page-level encryption and virtual machine isolation using AES. While these solutions offer robust protection, their integration is limited to specific hardware platforms, and their performance impact varies depending on workload characteristics.

Software-based approaches, such as Address Space Layout Randomization (ASLR) and memory segmentation, aim to prevent unauthorized access by randomizing memory layouts or restricting process-level access. However, these methods are ineffective against physical attacks like Cold Boot, where data remnants in RAM can be extracted post-shutdown.
Recent advancements in memory simulation have facilitated the evaluation of security mechanisms under controlled conditions. Ramulator 2.0, developed by CMU SAFARI [5], is a cycle-accurate DRAM

simulator that supports modular extensions for testing novel architectures and security protocols. It has been used to study RowHammer mitigation techniques, such as PARA and Graphene, and provides a flexible platform for integrating cryptographic plugins.

This work builds on these foundations by proposing a software-based AES encryption plugin integrated into Ramulator 2.0, coupled with an optimized memory scheduler to address the performance-security trade-off. Unlike hardware-centric solutions like TME or SME, our approach operates at the memory controller level, offering platform-agnostic flexibility. Additionally, by leveraging a custom scheduler, this study extends prior work to reduce encryption-related latency, providing a practical framework for securing memory systems against hardware attacks while maintaining acceptable performance.

III. The Proposed Model

This section introduces a software-based encryption framework aimed at enhancing memory system security against physical hardware attacks, with a particular focus on performance efficiency. The model operates at the architectural level and is implemented within the Ramulator 2.0 simulation environment, enabling real-time encryption and decryption of memory transactions. The following subsections provide a detailed overview of the model's architecture, implementation details, and the rationale behind design choices.

A. Model overview

The proposed model consists of two primary components: an AES encryption plugin and an optimized memory scheduler, referred to as `EncryptionScheduler`. The encryption plugin employs AES with a 128-bit key to secure data during read and write operations, ensuring that data stored in DRAM remains unreadable without the decryption key. To mitigate the latency introduced by cryptographic operations, the `EncryptionScheduler` optimizes the queuing and processing of memory access requests, prioritizing efficiency while maintaining security guarantees. The model is designed to operate between the CPU and DRAM, intercepting memory transactions at the memory controller to apply encryption and decryption seamlessly.

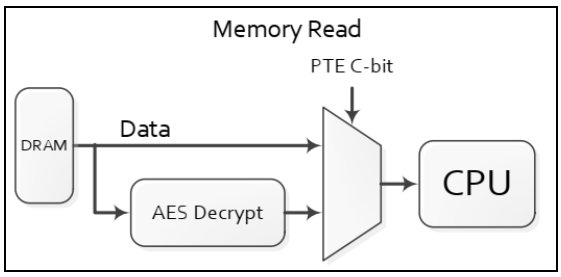

Fig. 1 Decryption on read.

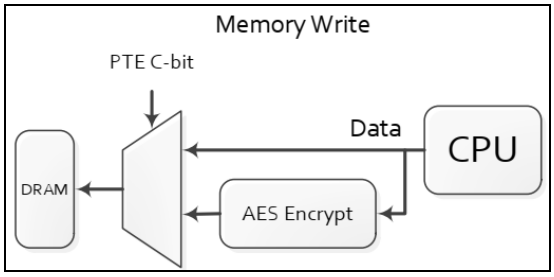

Fig. 2 Encryption on write.

The operation of the AES encryption plugin within the Ramulator 2.0 simulator is illustrated by the accompanying schemes above. In figure 1, data is retrieved from DRAM and passed through the memory controller, where a Page Table Entry Check-bit (PTE C-bit) determines whether the data requires decryption. The AES Decrypt module then processes the encrypted data, converting it into a readable format before it is delivered to the CPU. This workflow ensures that all memory read operations are secured transparently, aligning with the plugin’s design to intercept and process transactions in real-time. The Encrypt module works in the same way but in a reverse logical order (CPU->DRAM)

.

B. AES Encryption Plugin

The AES encryption plugin is implemented within Ramulator 2.0, leveraging its modular architecture to integrate cryptographic functionality into the memory controller. The plugin processes memory access requests (read and write) by applying AES encryption and decryption operations. The plugin structure includes:

- `aes_engine.h`: Defines the AESEngine class, a modular interface for AES encryption and decryption, supporting key sizes of 128, 192, or 256 bits. The class is integrated with simulator's controller framework to process memory transactions efficiently [22].
- `aes_plugin.h`: Provides the `create_aes_engine_plugin()` function, which serves as a factory for instantiating the AES encryption plugin, ensuring compatibility with the simulator's controller interface through a `std::unique_ptr` for automatic memory management [22].
- `aes_config.h`: Manages encryption parameters, such as the cryptographic key and policies for enabling encryption on write operations and decryption on read operations, read from a configuration file (`aes_config.txt`) [22].

The plugin is implemented in `aes_encryption_plugin.cpp`, processes data in 64-byte blocks. The encryption process is executed via the `encryptData()` method, which handles full and partial blocks, while decryption is performed similarly using `decryptData()` [22].

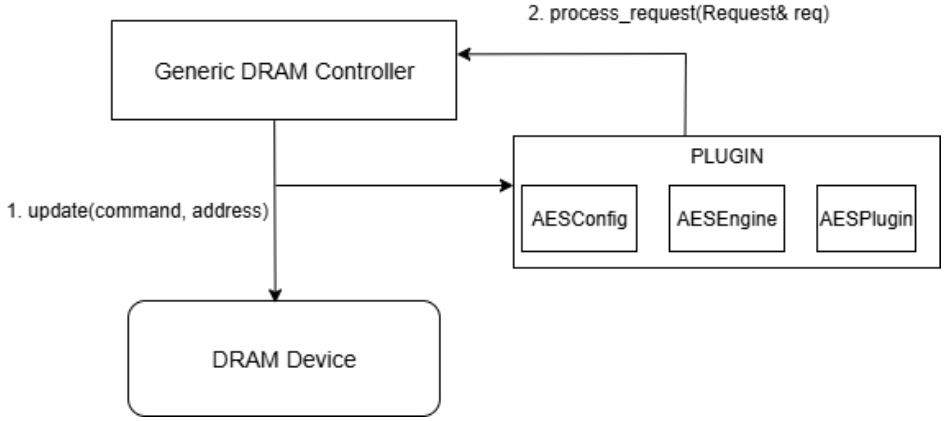

Fig. 3 Architecture of plugin integration into the simulator

Performance metrics, such as encryption cycles, are tracked using high-resolution timin `(std::chrono::high_resolution_clock)` to quantify latency overhead [22].

The plugin processes memory requests by intercepting them at the memory controller, applying AES encryption for write operations and decryption for read operations. For example, simulation logs demonstrate successful encryption and decryption for memory addresses, such as 0x1111 and 0x2112, with an average encryption latency of approximately 928.75 cycles per operation for a 128-bit configuration.

C. EncryptionScheduler

.

To address the performance overhead introduced by AES encryption, the proposed model incorporates EncryptionScheduler, a custom memory scheduler designed to optimize the handling of memory access requests. Since the decryption process takes more cycles and adds more latency, unlike traditional First-

Ready, First-Come-First-Serve (FRFCFS), EncryptionScheduler prioritizes requests based on their readiness and type (read or write), and gives priority to read instructions.
The scheduler operates as follows:

1. Request Queueing: Memory access requests are queued in a buffer, with high-priority requests (e.g., refresh operations) handled by the refresh manager to ensure memory integrity;
2. Request Processing: The scheduler decodes requests based on the current memory state, selecting the most suitable request to minimize latency.
   - *Between a read and write request*, the read requests are prioritized since they contain a decryption operation and they are considered as critical operations. Which means, that reads could block the processor, unlike the write operations which are more manageable (e.g., storing the writes in a buffer).
   - *If the requests are of the same type and readiness*, the scheduler applies FRFCFS to select the earliest request.

By optimizing request ordering, `EncryptionScheduler` reduces the average encryption cycles from approximately 750 cycles per operation to 450 cycles per operation and the average decryption cycles from approximately 900 cycles per operation to 500 cycles per operation.

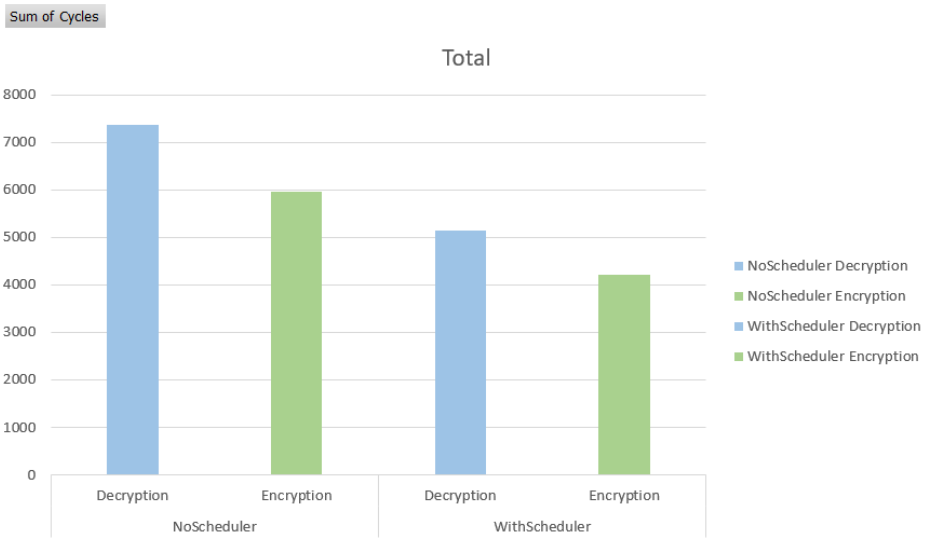

Fig. 4 Comparison of cryptographic latency cycles with and without Encryption Scheduler

The scheduler is integrated into Ramulator 2.0 via the `example_config_aes.yaml` configuration file, which specifies the memory hierarchy, controller settings, and encryption parameters. Simulation results, detailed in `results.csv` [22], show that EncryptionScheduler achieves a significant reduction, approximately 40%, in latency compared to configurations without the optimized scheduler.

D. Implementation Environment

The proposed mode is implemented using C++ within the Ramulator 2.0 simulator, running on an Ubuntu/WSL platform. The build process uses CMake to compile the AES encryption plugin dynamically as a shared library (`aes_encryption.so` [22]), ensuring integration with the simulator's infrastructure.

The simulation workflow involves loading a trace file (`aestest.trace`) and configuration file (`example_config_aes.yaml`) to execute memory transactions and collect performance statistics. The implementation leverages standard DRAM configurations, such as DDR4, and supports modular extensions for future enhancements.

IV. CONCLUSIONS AND FUTURE RESEARCH

A. *Key Findings*

This study demonstrates the efficacy of a software-based AES encryption plugin integrated into the Ramulator 2.0 simulator as a viable strategy for securing memory systems against hardware-based attacks. Simulation results indicate that the proposed model achieves robust data protection by rendering memory contents unreadable without the decryption key, effectively mitigating these threats. The AES encryption plugin, operating at the memory controller level with a 128-bit key, processes data in 64-byte chunks (comprising four 16-byte blocks). Performance evaluations reveal an average encryption latency of approximately 928.75 cycles per operation, with an overhead ranging from 1-5% for full-memory encryption, consistent with hardware-based solutions like Intel TME.

The introduction of the `EncryptionScheduler` significantly enhances the model's performance by reducing the average encryption cycles from 750 to 450 cycles per operation. This optimization, achieved through efficient request queuing and prioritization, demonstrates a substantial improvement in latency management, making the framework suitable for latency-sensitive applications.
These findings underscore the potential of software-driven encryption as a flexible, platform-agnostic approach to memory security.

B. *Suggestions for Future Research*

While the current study provides a solid foundation, several avenues exist for further exploration and enhancement. First, extending the AES plugin to support higher key sizes (e.g., 192 or 256 bits) could enhance security against advanced cryptographic attacks, though this may necessitate further optimization to mitigate increased latency.

Second, the `EncryptionScheduler`'s performance could be refined by incorporating machine learning techniques to dynamically adapt scheduling policies based on workload characteristics. This could involve training a model on simulation data to predict optimal request prioritization, potentially reducing latency further beyond the current 450 cycles per operation. Additionally, exploring multi-threaded or parallel processing of encryption tasks within the scheduler could address scalability challenges in multi-core systems.

Third, the model's applicability could be expanded by testing it across diverse DRAM technologies, such as DDR5, LPDDR5, and HBM3, to assess its robustness across modern memory architectures. Incorporating real-world hardware constraints, such as power consumption and thermal effects, into the

Ramulator 2.0 simulations would provide a more comprehensive evaluation of the framework's practical deployment.

These extensions would contribute to a more resilient memory security ecosystem, addressing emerging threats and evolving computational demands.

REFERENCES

[1] David A. Patterson, John L. Hennessy, "*Computer Organization and Design: The Hardware/Software Interface*", 2008

[2] David A. Patterson, John L. Hennessy, "*Computer Architecture: A Quantitative Approach*", 2011

[3] Ch. Paar, J. Pelzl, "*Understanding Cryptography*" (pp. 1 – 23, 55 – 82, 87 – 117, 331 - 352), 2010

[4] Tom St Denis, S. Johnson, Cryptography for Developers (pp. 139 – 202), 2007

[5] H. Luo, Y. C. Tuğrul, F. N. Bostancı, A. Olgun, A. G. Yağlıkçı, and O. Mutlu, "Ramulator 2.0: A Modern, Modular, and Extensible DRAM Simulator," arXiv preprint arXiv:2311.00001, 2023. Available: https://github.com/CMU-SAFARI/ramulator2.

[6] FIPS 197, Announcing the ADVANCED ENCRYPTION STANDARD (AES), 2001

[7] S. Deshmukh, R. Nair, Sh. Durugkar, S. Shinde, "Implementation of AES", IRJET, 2021.

[8] D. Kaplan, J. Powell, T. Woller, "AMD MEMORY ENCRYPTION", 2021.

[9] A. C. Mert, "Implementation Aspects of AES", TU Graz lecture slides, 2021.

[10] A. H. Tuch, A. Olmsted, "Physical Memory Attacks and a Memory Safe Management System for Memory Defense", New York University Tandon School of Engineering, Simmons University Boston, 2024

[11] D. J. Bernstein, "Cache-timing attacks on AES", 2005

[12] L. Biernacki, M. Z. Demissie, K. B. Workneh, F. A. Andargie, T. Austin, "Sequestered Encryption: A Hardware Technique for Comprehensive Data Privacy", IEEE, 2025

[13] O. Mutlu & J. S. Kim, "RowHammer: A Retrospective," IEEE Transactions on ComputerAided Design of Integrated Circuits and Systems, vol. 39, no. 8, pp. 1555-1571, 2020.

[14] V. Prutyanov, M. Romashikhin, Y. Vugenfirer, R. Soloyev, A. Romanov, "Impact of Memory Hierarchy on Memory Encryption Performance", IEEE Access, 2024

[15] O. Mutlu, J.Mexa, L. Subramanian, "The Main Memory System: Challenges and Opportunities", Carnegie Mellon University, 2015

[16] Intel Technology Brief (5th edition), "Memory Technology Evolution: an Overview of System Memory Technologies", 2005

[17] T. Austin, "Sequestered Encryption Primer", 2022

[18] Ch. Yan, B. Rogers, D. Englender, Y. Solihin, M. Prvulovic, "Improving Cost, Performance, and Security of Memory Encryption and Authentication", 2006

[19] T. Sh. Goo, "Intel Total Memory Encryption: Functional Verification and Performance Analysis", 2023

[20] https://www.intel.com/content/www/us/en/developer/articles/news/runtime-encryption-of-memory-with-intel-tme-mk.html

[21] Lamster, Lukas ; Unterguggenberger, Martin ; Schrammel, David et al. / Voodoo: Memory Tagging, Authenticated Encryption, and Error Correction through MAGIC. Proceedings of the 33rd USENIX Conference on Security Symposium. USENIX Association, 2024. S. 7159 – 7176

[22] https://github.com/aminatpwk/ramulator2-AES

[23] B. Stroustrup, The C++ Programming Language, 4th ed. Boston, MA, USA: Addison-Wesley, 2013.