Amplification of Errors by Encryption

Evgeny Manzhosov Simha Sethumadhavan
Department of Computer Science, Columbia University, New York, New York, USA
{evgeny, simha}@cs.columbia.edu

ABSTRACT

This paper shows how memory encryption amplifies the magnitude of errors miscorrected by the Error Correcting Codes. Moreover, we conducted a series of fault injections on programs with and without memory encryption and discovered that encryption increases the rates of silent data corruption and hangs.

1. INTRODUCTION

It is hard to imagine a modern enterprise-level CPU without reliability and security features. For example, every enterprise-level CPU today will offer both Error Correction Codes (ECC) [1,4] for main memory and memory encryption to guarantee user data confidentiality [2, 3, 6]. The typical approach taken by computer architects was to develop and apply security and reliability techniques independently, i.e., without considering the interactions between the two. Our paper studies the interaction between these system features - memory encryption and ECC, and reveals interesting results.

Those results are best illustrated with a simple example involving memory encryption and ECC, as shown in Figure 1. Let us assume a system with memory encryption and ECC, e.g., Hamming SEC-DED or Reed-Solomon, to improve memory reliability. In this system, upon data write, the CPU first encrypts the data into the ciphertext, then computes the ECC bits of the ciphertext, and, at last, stores both in the main memory. When the CPU requests data from memory, the ECC bits and ciphertext are checked for errors, and then the data is decrypted by the CPU. At first glance, it might appear that memory encryption, and ECC are completely orthogonal, and the addition of memory encryption into a system with ECC has no effect on reliability: the ability of an error correction code to correct errors is independent of whether the data is encrypted or not. It turns out, however, that reliability is affected.

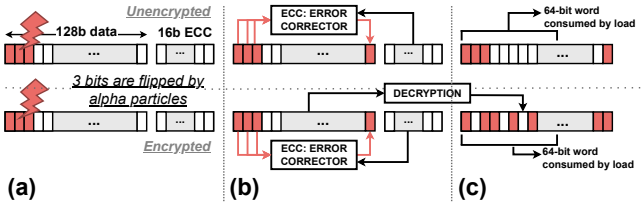

Figure 1: The lower pane of the figure shows how the bit diffusion of encryption algorithms affects the magnitude of the miscorrected errors compared to memory without encryption. If corrupted with a 3-bit error (a), a data word with ECC might be miscorrected by the ECC (b) and amplified by encryption into an error of higher magnitude (c) due to bit diffusion during decryption.

This effect of encryption on reliability shows in cases when errors corrected by the ECC are outside of the assumed fault model, i.e., triple-bit errors for the SEC-DED code. In this case, if the data is encrypted, such errors will cause more bits to be erroneous after the decryption due to the diffusion property of the encryption algorithms. For instance, a 3-bit error on a 16-byte block with Reed-Solomon would result in (at most) four flipped bits without encryption as opposed to 64 flipped bits in the same encrypted 16-byte block after decryption. Thus, workloads that run on systems with encryption may experience degradation of reliability guarantees.

This paper provides insight into how encryption-diffused errors, affect the reliability of CPU workloads. To this end, we conducted a series of fault injection campaigns, and we see that all studied workloads experience significant reliability degradation. For example, CPU-based workloads experience up to 6.92× more Silent Data Corruptions (SDC) with encrypted memories compared to workloads executed on a system without encryption.

2. METHODOLOGY AND RESULTS

Throughout this study, we fix the data granule of interest to be 16 bytes for two reasons. First, this is the block size of AES cipher often used in various memory encryption schemes [3, 6], and second – this is the codeword size of ECC schemes used in commercial settings, such as Reed-Solomon-based ECC used in AMD CPUs [1]. We analyze two types of systems: `baseline` and `secure`. Both have ECC; however, only the `secure` system has memory encryption. Furthermore, we assume that during the lifetime of an application, the data is first encrypted (only in `secure`) with AES, then encoded with ECC, and only then it is written to the main memory. Upon memory read, the same procedure is performed in reverse: check for errors and then decrypt.

2.1 Fault Injection Process

The backbone of our fault injection infrastructure is CRIU (Checkpoint and Restore In Userspace) – a utility that checkpoints the program's memory state to disk and restores it later, resuming execution. In a nutshell, we checkpoint a program at the time t_{inj}, randomly pick an injection address A_{inj} in the checkpoint, and then resume execution, effectively modeling a case of a program with corrupted memory.

Injection Times and Addresses We use the `time` utility in Linux to measure the runtime of all the workloads to ensure that our fault injection times, t_{inj}, are uniformly sampled and cover the entire program's lifetime. To inject the fault at time t_{inj}, we use the `sleep` utility to delay CRIU check-

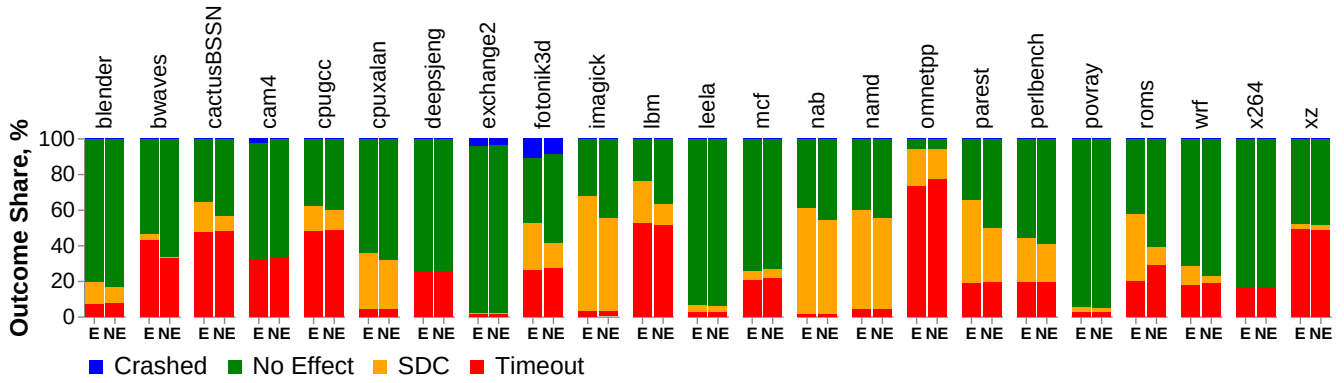

Figure 2: Results of the Fault Injection experiments for the SPEC'17 benchmarking suite. Each bar shows the share, in per cents, of each outcome category: `Crashed` in blue, `Timeout` in red, `SDC` in orange, and `No Effect` in green. For example, only four programs crashed due to Segmentation Fault – `cam4`, `exchange2`, `fotonik3d`, and `wrf`.

point command by t_{inj}. Once the program is checkpointed, we determine the size of the program's memory state and pick a random injection address A_{inj} aligned to the nearest 16B boundary. This way, we ensure that both the injections' time and address are randomly sampled from the program's memory state. Moreover, we use the same checkpoint at t_{inj} for both the `baseline` and `secure` system to guarantee that both injection trials are consistent and represent the effects of the same fault in memory, with the only difference being the amplification of the error by encryption, as described next.

Memory Errors Generation We profile the Reed-Solomon-based ECC scheme to generate memory errors and find error patterns miscorrected by ECC. To do so, we randomly generated 1M of double-chip error patterns (as those are beyond the fault model of single-device correct ECC), from which we kept those detected by ECC as correctable errors. We randomly pick one of those error patterns, e_{inj}, for both `baseline` and `secure` systems and use it to corrupt the checkpoint at address A_{inj}. However, for the `secure` system, we do not use e_{inj} as is; instead, we use it to generate an encryption-diffused error as a four-step process: (1) we read 16B of data from the checkpoint at address A_{inj}, (2) we encrypt this data with AES, (3) we corrupt the resulting ciphertext with e_{inj}, (4) decrypt the ciphertext and update the checkpoint with data corrupted by diffused error. This way, for the `secure` system, the checkpoint has an error amplified by the encryption. We use the same checkpoint for both injections to ensure that both experiments model the same error, at the same time, t_{inj}, address A_{inj}, and error, e_{inj}.

Outcome Classification For each of the injections, we log the time, address, the memory region the address A belongs to, e.g., stack, heap, etc. In line with prior work [5], we assume that if program's execution is longer than 3x of it remaining error-free execution time, it entered bad state, and we terminate it with `timeout` utility by sending a `kill` signal. We later categorize injection outcomes into following categories:

- Timeout – program execution is longer than 3× its normal execution time after the injection.
- Crashed – segmentation fault during execution.
- Silent Data Corruption (SDC) – program finished in time, but the output differs from the error-free execution.
- No Effect – program finished in time and the output matches error-free case.

2.2 Experimental Setup

We use the methodology described in [7] to get 95% confidence level with 2.1% error margin with 2000 injections for each workload. To study general-purpose workloads, we use SPEC-2017 CPU benchmark suite updated to v1.1.9 compiled with g++ 11.3.0 with `-O3` optimization level under Ubuntu 22.04 for aarch64. To allow for reasonable execution times, we use train inputs for the benchmarks. We conducted the experiments on the Ampere Altra server with 256GB RAM and 160-core aarch64 CPU.

2.3 Experimental Results

Figure 2 shows the results of the fault injection campaign on the SPEC'17 workloads categorized into Crashed (blue), Timeout (green), SDC (orange), or No Effect (green). We see from the results that with memory encryption for some programs, SDC rates increase dramatically, e.g., `bwaves` with 6.92×, `roms` with 3.63×. On the other hand, Timeouts are application dependent, e.g., `bwaves` has about 30% more hangs with encrypted memory, while `roms` has about 45% fewer timeouts with encrypted memory. These results show that adding encryption could cause significant reliability degradation (SDC) and availability (Timeouts) guarantees, highlighting the need for security-reliability co-design of the new generation of reliability features for secure processors.

3. CONCLUSION

In this paper, we reported that adding memory encryption will degrade the reliability guarantees of the system. To emphasize the scope of the effect, we analyzed SPEC'17 workloads in the settings of encrypted memory. We saw that adding encryption increases rates of silent data corruption and may cause a significant increase in the program's hangs. Given current adoption trends of privacy and security technologies, these findings would be of interest to the architectural community.

ACKNOWLEDGMENTS

This work was partially supported by Qualcomm Innovation Fellowship and Google Research Gift.

REFERENCES

[1] "BIOS and Kernel developer's guide (BKDG) for AMD family 15h models 00h–0fh processors," Advanced Micro Devices, Inc., Tech. Rep., January 2013, rev. 3.14.

[2] "ARM architecture reference manual supplement for ARMv9-A architecture profile." ARM Limited, Tech. Rep., 2021, Rev. A.d.

[3] P.-C. Cheng, W. Ozga, E. Valdez, S. Ahmed, Z. Gu, H. Jamjoom, H. Franke, and J. Bottomley, "Intel TDX Demystified: A Top-Down Approach," Mar. 2023, arXiv:2303.15540 [cs]. [Online]. Available: http://arxiv.org/abs/2303.15540

[4] L. Cojocar, K. Razavi, C. Giuffrida, and H. Bos, "Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks," in *2019 IEEE Symposium on Security and Privacy (SP)*, San Francisco, California, USA, 2019, pp. 55–71.

[5] M. Kaliorakis, D. Gizopoulos, R. Canal, and A. Gonzalez, "MeRLiN: Exploiting dynamic instruction behavior for fast and accurate microarchitecture level reliability assessment," in *44th Annual International Symposium on Computer Architecture (ISCA)*, 2017, pp. 241–254.

[6] D. Kaplan, J. Powell, and T. Woller, "AMD Memory Encryption," Advanced Micro Devices, Inc., Tech. Rep., 2021.

[7] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation & Test in Europe Conference & Exhibition*, 2009, pp. 502–506.