To appear in MICRO-2024. Authors Preprint.

Polymorphic Error Correction

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

***Abstract*—In this paper, we propose a new memory error correction scheme, Polymorphic ECC, based on a novel idea of *redundancy polymorphism* for error correction. With redundancy polymorphism, we can use the check bits, i.e., parity bits in traditional ECC, to correct errors from different fault models. For example, the error correction procedure will use the same redundancy value for single-bit errors, double-bit errors, Chip-Kill, and others. As a result, Polymorphic ECC corrects more errors than traditional codes, which typically target a single fault model or require multiple redundancies for multi-fault model support, leading to higher storage overheads. Our construction is very compact, allowing us to embed an *inlined* cryptographic message authentication code (MAC) with each cacheline, ensuring data integrity and near 100% error detection without needing any extra storage. The MAC, further permits iterative correction among the many supported fault models. In the paper, we show that the novel combination of redundancy polymorphism with iterative correction, corrects errors due to fault models not covered by traditional codes and guarantees data integrity with up to 60-bit MACs while using 64-byte cachelines and standard 40-bit DDR5 memory channels.**

***Index Terms*—Memory Error Correction, Security, Reliability, Polymorphism.**

I. Introduction

Today, Error Correction Codes (ECC) are critical for offering acceptable levels of system reliability. Their effectiveness, however, is increasingly challenged by emerging security threats [14] and modern defense mechanisms [74]. Specifically, rowhammer faults [64], [71] and extremely tight design margins exacerbate the reliability concerns of modern memory systems. At the same time, new security defenses, such as memory encryption [4], [11], [40], further strain program reliability since faults that escape traditional ECC mechanisms result in errors that diffuse and propagate more widely. Thus, in this paper, we present a novel ECC mechanism designed to address the dual challenges of ensuring both the reliability and security of modern memory systems.

We describe a novel error correction scheme, Polymorphic ECC, which significantly enhances error detection and correction capabilities compared to current state-of-the-art codes while staying within the bits budgeted to support traditional ECC. Because of how it is constructed, Polymorphic ECC permits at least a 40-bit MAC to be stored with each 64-byte cache line to detect any pattern of bit corrections with a probability of $1 - 2^{-40}$ with no additional area overhead. Further, Polymorphic ECC also permits storing 11 redundancy bits per 64 data bits, allowing it to correct any single-bit error, two random bit errors, eight-bit errors within naturally aligned eight-bit symbols, and more, again, at the cost of no additional storage. In contrast, a state-of-the-art Reed-Solomon code, for the same storage budget as Polymorphic ECC, can only correct symbol-aligned errors [61], with a ∼7% chance of miscorrecting other corruptions. This limitation becomes even more pronounced when ECC bits used for traditional error correction mechanisms are shared for security features such as memory safety tags [58], [74], further reducing the efficacy of conventional codes [51].

Traditional constructions of ECC have redundancy information crafted (mathematically) to work with only one fault model. However, our construction uses residue codes, where the redundancy information is the remainder (residue) left over after dividing the data word by a known constant [26]. With this approach, we gain two advantages on parity-based codes. First, we gain *redundancy polymorphism*, i.e., the ability to map the same residue value to errors due to multiple fault models. Second, as we discovered, the cardinality of those mappings is much smaller than that of those made with traditional parity, making the search space for errors much smaller and error correction much faster.

For example, consider a 32-bit word protected by 4-bit parity, where one bit protects 8 bits of data, i.e., the first parity bit protects bits 0 to 7, the second parity bit protects bits 8 to 15, etc. Suppose we know there was a two-bit error in the data, and we need to correct it through search. When we recompute the parity bits, we can have two outcomes: (1) two parity bits do not match, which will indicate corrupted bytes, bounding the search space to 64 error candidates, and (2) parity bits match, which means that we have to try all possible 2-bit errors that are in the same byte or $4 \times \binom{8}{2} = 112$ error candidates. In the same settings, Polymorphic ECC, as we show in the paper, needs at most 18 and 10 error candidates, respectively, resulting in a significant reduction of search space.

Conceptually, Polymorphic ECC works in two steps: (1) error detection via MAC check and (2) error correction via iterations. The error detection is done by comparing an inlined MAC with a new one recomputed from the data, and if those MACs do not match, we declare an error. In this case, we begin the iterative correction process. First, we use codeword residue to derive probable errors for the first fault model, say, 1-bit errors. Then, for each iteration, we use one of those error candidates to correct the error, and the outcome of each correction trial is verified by MAC check against inlined MAC. Suppose none of the error candidates of the first fault model corrected the error and passed MAC verification. In that case, we try the second fault model, say random 2-bit errors; we again use the *same* codeword residue as before to derive error candidates for the 2-bit errors and use them for correction. This iterative correction continues until MACs match for one error candidate or exhaustion of fault models, in which case we detect an uncorrectable error.

To better understand our solution, let's walk through a simple example. Imagine we are storing the number 100 in memory, and for error correction, we use a constant of 3. Under Polymorphic ECC, we store two key pieces of information: a hash of the number 100 (let's assume this is a 3-bit hash, and the result is 5) and the remainder (or residue) when 100 is divided by 3, which is 1. Suppose a bit flip occurs, and instead of 100, we read 101. When we recalculate the hash of 101, we get a different value, say 6, detecting an error. To correct the error, we compute a new residue value of 101 divided by 3, getting 2. Then, for each fault model, we select error candidates that are mapped to a residue value of 2 and use them for error correction. We continue this way until we either correct the error or exhaust all possible fault models we want to handle. The feature of redundancy polymorphism enables the code to support multiple fault models with a small number of redundancy bits.

To intuitively understand the benefits of such a code, consider a memory under Rowhammer conditions. Here, Rowhammer conditions could mean faults that are caused by an attacker or unintended pathological conditions. A foolproof approach to detect Rowhammer is storing a MAC of memory at some granularity, such as cacheline, in a decoupled part of the DRAM. This simple solution has some drawbacks. First, this MAC is designed to be orthogonal to the error correction code present in this system, which already provides some — weak — detection capability. Thus, this solution makes inefficient use of a resource that is already present. Second, fetching the MAC requires another memory access, costing bandwidth and energy. Third, storing the MAC increases the memory footprint and execution time of the program, both of which increase the vulnerability to benign faults. Finally, the MAC-based solution does not offer any correction guarantees, which may be necessary for high-availability systems. Furthermore, the error patterns induced with unintentional Rowhammer may not fit the traditional fault models designed around random bit flips or typical failures like one chip of a DIMM failing. In contrast, Polymorphic ECC circumvents these drawbacks by storing the MAC *inline and alongside* the data, reducing bandwidth and energy cost, working synergistically with ECC residue bits to offer correction, which is good for an unintentional Rowhammer, and finally, avoiding storing additional metadata which further improves reliability by reducing memory vulnerability.

Concretely, in this paper:

- We propose Polymorphic ECC, an ECC scheme that can use a single set of ECC bits to correct errors due to multiple fault models. Specifically, each error correction attempt will be made by treating existing ECC bits as if those were encoded according to a different fault model. For example, Polymorphic ECC can use cacheline's ECC bits to correct any error due to random double-bit, double-bounded, or single-device faults, providing better DDR5 reliability than industry-standard ChipKill.
- We show that Polymorphic ECC improves system security with support for long MACs. Since Polymorphic ECC uses fewer ECC bits than other ChipKill schemes, the freed space is reused to embed cryptographic MACs, providing data integrity. Specifically, Polymorphic ECC allows at least 42% longer MACs than Intel TDX [11].
- We show how increased fault coverage of Polymorphic ECC improves system reliability against rowhammer. Polymorphic ECC corrects rowhammer-induced errors that are not correctable by Reed-Solomon or Unity ECC schemes. Thus, a system with Polymorphic ECC spends more time doing useful work than restarting due to uncorrectable errors.

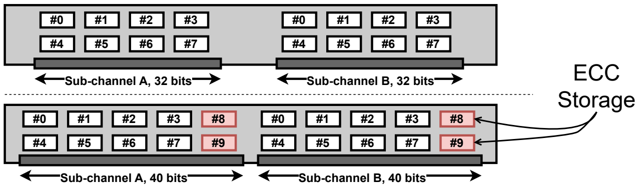


Fig. 1: Top: DDR5 memory module with eight x4 DRAM devices per 32-bit sub-channel. Bottom: DDR5 memory module with two x4 devices for ECC storage and 40-bit sub-channels.

II. Background

This section provides a brief background information relevant to this work.

A. Modern Main Memory Organization

The latest generation of DRAM, DDR5, was introduced in 2022 and uses 64-bit-wide memory modules with two 32-bit memory sub-channels [34]. As a result, to provide cacheline worth of data, i.e., 512 bits assuming 64 byte cachelines, each sub-channel performs 16 data transfers: $16 \times 32 = 512$, and the IO width of DRAM chips determines the total number of devices on a memory module. For example, with 4-bit-wide DDR5 DRAM chips (often called x4 DRAMs), each memory sub-channel will have eight devices, to a total of 16 per module, as shown in the top of Figure 1. For the server-class systems, memory modules come with additional DRAM chips to enable fast access to ECC check bits, i.e., redundancy bits, as shown at the bottom of Figure 1, resulting in a wider 40-bit interface.

B. Fault Models and Error Correction

DRAM Fault Models. In the context of memory error correction, fault models capture how various physical failures within DRAM devices affect stored data, e.g., a stuck-at-1 bit on DRAM IO or in the storage array. Both can manifest as single-bit errors but require separate fault models, as failed IO can corrupt the data on each memory read/write, while a failed array bit affects only one word. Another example is a complete device failure fault model that can result in all the

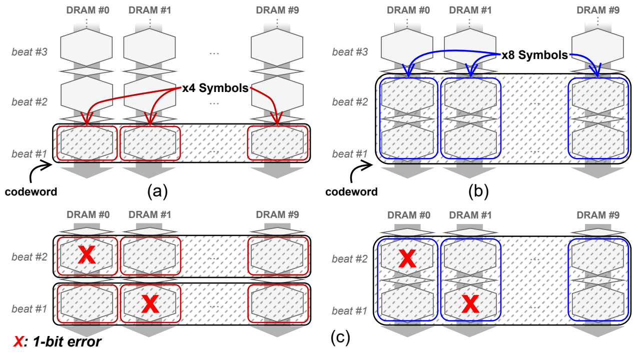


Fig. 2: Example of SSC configuration vs. symbol size for memory with x4 DRAMs: (a) 4-bit symbols, (b) 8-bit symbols, and (c) coverage of single-bit errors by codewords with 4-bit (left) and 8-bit symbols (right), where 8-bit configuration results in double-symbol uncorrectable error vs. two codewords with 4-bit symbols with correctable single symbol errors.

data read from the device being incorrect. This fault model is used to design the Single DRAM Device Correct (SDDC) error correction codes, colloquially referred to as ChipKill codes, which are standard in server-class CPUs today.

ECC Design Considerations. The commonly used codes to provide ChipKill guarantees are symbol-based codes, which map data read from every DRAM chip into distinct symbols. With enough redundancy, those codes can correct several symbols with errors, e.g., code that corrects only one symbol is called a Single Symbol Correcting (SSC) code.

To illustrate the design space of symbol size, device width, and error coverage for SSC-class code, we will use the illustration in Figure 2 that shows how data transmitted via memory interface across multiple beats is grouped into symbols, and how those symbols are mapped to DRAMs. We see in Figure 2(a) that for the SSC code, the size of the symbol has to be at least the width of the DRAM device; otherwise, upon device failure, more than one symbol will be corrupted, and the code won't correct the error. On the other hand, Figure 2(b) shows how the symbol size can be a multiple of the device width, which results in one symbol containing the data across multiple beats, e.g., an 8-bit symbol stores two beats worth of data from a x4 DRAM chip. In the worst-case scenario of chip failure, both configurations guarantee data recovery and result in SDDC code.

However, if we want our ECC scheme to tolerate single-bit errors on every memory read as well, which are more common than complete device failures, the configuration from Figures 2(a) and (b) perform differently. Figure 2(c) shows the case of two single-bit errors that happen in the first and the second beat. In this case, the code with 4-bit symbols will correct both errors as they are detected as single-symbol errors in different codewords. However, the code with 8-bit symbols will detect a two-symbol error, which is beyond the correction abilities of SSC code. Thus, while both codes handle the worst-case scenario equally well, the performance of the first code in the common case is better. To put it in the context of modern systems, AMD uses code design with 8-bit symbols written across two beats into x4 devices [1] as opposed to Intel's SDDC, which uses 16-bit symbols [69].

MUSE ECC. In our work, we build upon the ideas of MUSE ECC [51], which provides SDDC guarantees. As shown in Eq. 1, MUSE ECC's codewords, C, are constructed by multiplying the data, D, with a constant integer multiplier M that is selected using a search procedure that maps remainders for each error pattern of interest. When the codeword C is read from memory, it is checked $modM$, and if the resulting remainder R is not zero, the error is detected. To correct the errors, MUSE ECC checks a pre-existing map, MAP, of all the errors and their remainders for an error E matching the remainder R and uses it to correct the error.

$$\begin{gathered} C = D \times M \\ R = C \bmod M \\ D_{corrected} = (C - MAP(R))/M \end{gathered} \tag{1}$$

To generate the MAP, MUSE ECC represents bit-flip as an integer power-of-two, i.e., flipped bit b_i has value 2^i. Thus, MUSE differentiates between 0→1 and 1→0 bit flips as those have different error integers, i.e., 2^i and -2^i, respectively. Similarly, symbol errors are sums of power-of-two integers aligned to specific symbol boundaries. For example, if the value of the third symbol in the codeword with 8-bit symbols increases by 3, its error integer is $(2^1 + 2^0) << 16$.

To correct errors, MUSE ECC picks a multiplier M so that all the errors in the fault model have a unique remainder, and this error-remainder mapping is used for error correction. With this construction, MUSE ECC uses fewer bits of storage than similarly configured Reed-Solomon (RS) codes. However, unlike RS codes, MUSE ECC is limited to 4-bit symbols because 8-bit or 16-bit symbols need multipliers that need more storage than common ECC budgets of 12.5% [33] and 25% [34]. Moreover, because MUSE ECC uses 4-bit symbols, it needs to use two memory channels, e.g., 80-bit vs. 40-bit with DDR5 memory, potentially reducing memory parallelism. In Section V, we describe how Polymorphic ECC addresses those limitations of MUSE ECC. Namely, SDDC Polymorphic ECC has no lookup tables, allows for 8-bit and 16-bit symbols, and works with standard 40-bit memory channels.

C. Memory Encryption

Memory Encryption has become a standard for many enterprise use cases such as medical records or financial transactions. Encrypting the data makes it look like a collection of random bits with no discernible patterns. This property is called diffusion [65], and its main function is to hide correlations between the original data and the ciphertext. To encrypt the data, (symmetric) encryption algorithms, substitute and permute the data based on a (secret) key and the process can be reversed during decryption. Due to bit diffusion, a single bit flip in the encrypted text can change the decrypted text significantly, often flipping about half of the bits. Today, memory encryption is used to guarantee data confidentiality

TABLE II: Misdetection Rates (%) for Out-of-Model Errors

Code	Number or Errors†							Average
	2	3	4	5	6	7	8	
`Hamming(72,64)`	0	75.9	0	67.9	0	62.5	0	**29.5**
`Reed-Solomon`	6.3	7.0	7.0	7.0	7.0	7.0	6.9	**6.9**

† bits for Hamming code, bytes w/ bit flips for Reed-Solomon

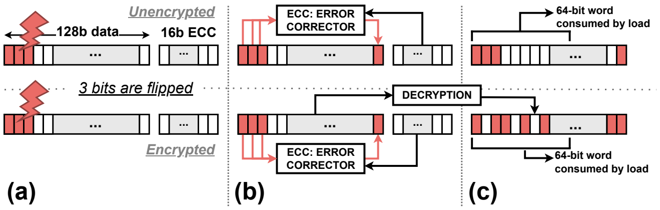


Fig. 3: The lower pane of the figure shows how the bit diffusion of encryption algorithms affects the magnitude of the miscorrected errors compared to memory without encryption. If corrupted with a 3-bit error (a), a data word with ECC might be miscorrected by the ECC (b) and amplified by encryption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

[4], [52] or ensure isolation of virtual machines in cloud settings [11], [40], and is based on the AES cipher [19].

III. Motivation

In this section, we introduce the notion of Out-of-Model Faults and show how they affect system reliability once the memory or computation is encrypted.

A. *Out-of-Model Faults*

Since a more robust ECC requires more storage for redundancy, ECC are designed for the most common faults. We refer to the faults that generate errors covered by the ECC as In-Model Faults and those whose errors are not covered as Out-of-Model Faults. For example, for the single error correcting and double error detecting (SEC-DED) ECC, the single and double-bit errors are In-Model, while triple-bit errors are Out-of-Model because the SEC-DED ECC cannot consistently correct those errors [29].

To illustrate this point, we profile ECC schemes and show how some Out-of-Model Faults are detected as either In-Model Faults or Out-of-Model Faults. Table II summarizes the profiling results for Hamming SEC-DED and Reed-Solomon ChipKill codes configured as in Figure 2(b). For Hamming code, 75.9% of triple-bit errors will be misdetected as single-bit errors and miscorrected into four-bit errors. Reed-Solomon code misdetects about 6.3% of double-chip errors as single-chip errors. Those results are similar to rates measured by Cojocar *et al.* in Intel and AMD CPUs [14]. Errors that are not miscorrected, e.g., 26.9% of triple-bit errors for Hamming code, will be classified as detectable uncorrectable errors. Thus, ECC schemes may misclassify the errors caused by the Out-of-Model Faults on a case-by-case basis, leading to a lack of consistency in their treatment. We discuss the difficulties with the estimation of Out-of-Model Faults in Section VII-A.

B. *Reliability Impact of Out-of-Model Faults*

To illustrate how Out-of-Model Faults affect the reliability of applications, we conduct a series of fault injections of Out-of-Model Faults into the general purpose (SPEC'17) and inference (image classification) workloads. We describe the fault injection setup and the methodology in Section VII-B.

Encryption-amplified Errors. These errors happen in systems with encrypted memory. In those systems, the data is first encrypted into ciphertext, then the ECC is applied, and ciphertext is written to memory. When the ciphertext is read back, first, the ECC is checked, and then the data is decrypted. At first glance, it might appear that memory encryption is completely orthogonal to ECC, and the addition of memory encryption into a system with ECC does not affect reliability: the ability of an error correction code to correct errors is independent of whether the data is encrypted or not. However, the orthogonality breaks once we consider Out-of-Model Faults. The ECC often miscorrects errors due to those faults, leaving the ciphertext corrupted before decryption, whose bit diffusion property will amplify the corruption in the plaintext data. For instance, some 3-bit errors in a 16-byte block with RS will be miscorrected and later amplified by AES into 64-bit errors, as shown in Figure 3. Thus, systems with memory encryption may experience degradation of reliability guarantees.

General Purpose Workloads. Figure 4 shows the results of the fault injection campaign on the SPEC'17 workloads categorized into Crashed (gray), Hang (orange), Silent Data Corruption (SDC, red), or No Effect (green) with encrypted (E) and not encrypted (NE) memory. We see from the results that with memory encryption (E) for some programs, SDC rates increase dramatically, e.g., `bwaves` with 6.92×, `roms` with 3.63×. No application showed reduction in SDC with encrypted memory. Hangs, on the other hand, are program dependent – `roms` has about 45% fewer Hangs with encrypted memory, while `bwaves` has about 30% more. These results show that adding encryption could cause significant degradation of reliability (SDC) and availability (Hangs) guarantees, highlighting the need for security-reliability co-design of the new generation of reliability features for secure processors.

ML Inference. The histograms in Figure 5 show the results of fault injection into the inference workloads with encrypted memory (a) and fully homomorphic encryption (b). Both figures show the average Top-1 inference accuracy changes due to Out-of-Model faults. We can see from the figures that fewer inferences remain accurate as a result of encryption-amplified errors. For example, Figure 5(a) shows that due to encryption, the share of inaccurate inferences increases (blue bars). Only 1079 injections have near baseline accuracy compared to 1286 without memory encryption, a decrease of 16%. Moreover, with encrypted memory, the number of failed inferences increased from 196 to 234 (+19%). Similar results are observed for inference with homomorphic encryption: 18.5% of inferences experience more than a 10% drop in

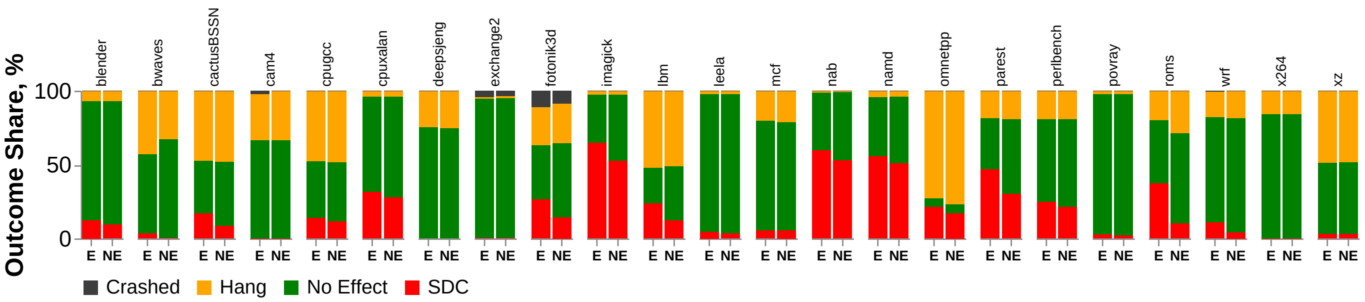


Fig. 4: Results of the Fault Injection experiments for the SPEC'17 benchmarking suite. Each bar shows the share, in per cents, of each outcome category: `Crashed` in gray, `Hang` in orange, `SDC` in red, and `No Effect` in green.

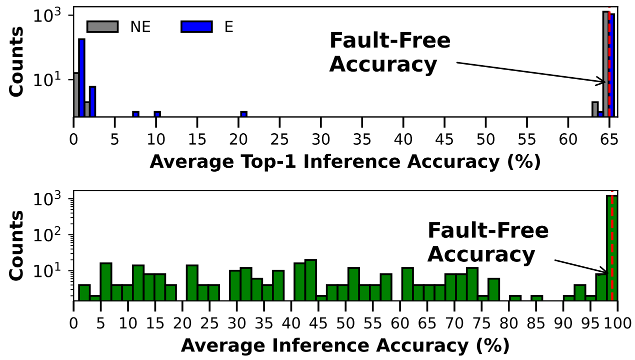


Fig. 5: Histogram with the accuracy distribution for (a) Mobilenet v2 classification network Inference and (b) Cryptonets [22] Inference with Full Homomorphic Encryption with injected Out-of-Model Faults. Y-axis is log scale.

accuracy. Given the nature of applications first to use FHE, e.g., inference on healthcare data, inaccurate data classification can pose a high risk to the well-being of patients and the quality of medical treatment, stressing the need for error mitigation techniques expanded to FHE systems as well.

Summary. To summarize, introducing encryption techniques can degrade a system's reliability guarantees due to the disastrous combination of miscorrected errors and diffusion of bits. Given current adoption trends of systems with encrypted memory and confidential computing [11], [25], Out-of-Model Faults pose a serious concern that needs to be addressed.

IV. Our Solution for Multi-Fault-Model Support

Our goal in this work is to reduce the number of errors due to Out-of-Model Faults, and we achieve that by security-reliability co-design. We observe a strong demand for private and secure computing that has begun being addressed by the industry, e.g., Intel SGX [52] and TDX [11], which provide cacheline data integrity guarantees via cryptographic MACs. From the memory reliability point of view, n-bit MACs provide us with near-perfect error detection probability of $1-2^{-n}$, which is far superior to standard ChipKill-level ECC.

Given near-perfect error detection, one solution for strong error correction is to iteratively try and fix all the errors until one of the fixes results in a MAC match. Variations of this approach were taken by a number of prior works [20], [31], [38], [62], [70]. However, the error search space is very large; thus, either the correction time will be prohibitively long [38], or the search space has to be limited to something reasonable such as SDDC [20], [31], [62], [70], which fails to meet our initial goal of covering more errors due to Out-of-Model faults. An alternate approach could be to use clever codeword organization [36], [42], [72], but those either leave no space for security metadata such as MACs or fail to cover some errors due to the symbol alignment limitations (Section II-B). An alternative to both of those could be using multiple ECCs simultaneously, i.e., one per fault model, which will lead to storage overheads beyond the 25% provided by the DDR5 standard. Hence, we face an interesting challenge: *How can we design a multi-fault-model ECC without prohibitive storage or runtime overheads?*

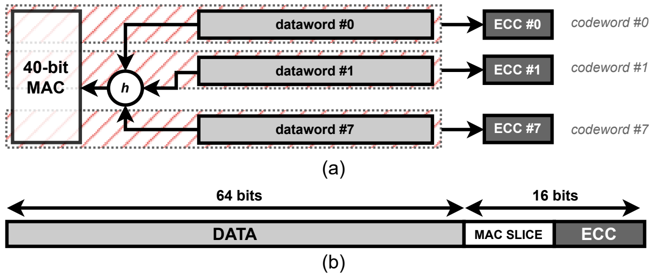
In this work, we address this problem with security-reliability co-design and develop a novel memory error correction scheme. Our approach, Polymorphic ECC, uses secure keyed MACs for error detection and iterative error correction to correct errors due to various fault models. Polymorphic ECC poses no restriction on the MAC itself, and thus, any MAC satisfying the security constraints of the system can be used, preferably with high resistance to collisions, to improve error detection capabilities. As a result, with Polymorphic ECC, fewer errors are categorized due to Out-of-Model Faults, improving system reliability guarantees.

V. Design of Polymorphic ECC

In this section, we describe the construction of Polymorphic ECC for DDR5 memories and how remainder polymorphism allows for wider symbols and correction of errors due to different fault models.

A. Polymorphic ECC overview

The core of Polymorphic ECC is strong error detection via MAC per cacheline. There are two possibilities to protect data with MAC and ECC: (1) ECC bits derived from data, both protected by the MAC, and (2) first, the MAC is computed from the data, and then ECC protects both. The downside of the first approach is that errors in MAC are not correctable, resulting



(a)

(b)

Fig. 6: (a) The MAC is computed on 64B of data, and distributed across eight codewords, whose ECC bits are computed on respective data and MAC slice. (b) Codeword structure of Polymorphic ECC: data (gray), slice of MAC (white), and redundancy bits (dark gray).

in high rates of detectable but uncorrectable errors. With the second approach, the bits of MAC are equally distributed among the codewords and protected by ECC along with data, as shown in Figure 6(a), converting previously uncorrectable errors into correctable ones. Figure 6(b) shows an example of Polymorphic ECC configured with 8-bit symbols, where each codeword stores 64-bit data and 16 bits of MAC and ECC, and the multiplier value determines how many of the available 16 bits are ECC bits. For example, the smallest multiplier with 8-bit symbols is 511, leaving 56 bits per cacheline for MAC. Next, we describe code construction, the error correction process, and an optimized version of remainder-to-error mapping.

B. Code Construction and Remainder Aliasing

Multiplier Search Procedure. A pseudocode implementation outlined in Algorithm 1 checks if a multiplier M can define an instance of Polymorphic ECC with N symbols of S bits each. If the multiplier defines a code, the algorithm will also compute the aliasing degrees for each remainder. For each symbol in the codeword, the code computes its bit offset in the codeword (line 2) and then computes a list of integer values for all the errors of that symbol (lines 3-4). Then, for each of the error integers, their remainder modulo M is computed (lines 5-7) and adds the result to the set of all symbol remainders, REMS (line 8). We compute the remainder for positive and negative error integers to account for both directions of bit flips (see Section II-B) and update the counts for their aliasing degrees (line 9). We terminate the check early if we do not have enough unique remainders to cover all errors in symbol (line 10). Otherwise, we continue until all symbols are checked (line 12). We return `True` and the histogram of aliasing degrees for a multiplier that has enough remainders in each symbol. For example, multiplier $M = 511$ has enough remainders to define a 10-symbol code with 8-bit symbols.

Remainder Aliasing and Error Candidates. As we see from the pseudocode, unlike MUSE ECC, which relies on remainder uniqueness, Polymorphic ECC allows remainder aliasing between error integers in different symbols because the remainder value is not used to localize the error. Thus, in Polymorphic ECC, one remainder can correspond to several error integers in different symbols, and we call this number remainder's *aliasing degree*. For example, for multiplier $M = 511$, the remainder $R = 1$ has aliasing degree of ten and is mapped to error integers: 1, 512, 1023, etc.

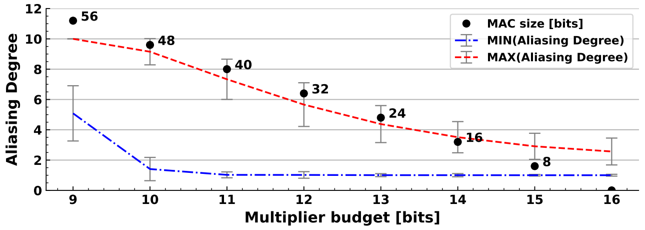


Fig. 7: The trade-off between multiplier size, average min/max aliasing degree per multiplier, and MAC size (8-bit symbols).

Furthermore, we make several important observations. First, with remainder aliasing, we need smaller multipliers than MUSE ECC to define a code because we don't have to have a one-to-one mapping between all the errors in the codeword, only within one symbol. Thus, Polymorphic ECC is more storage efficient than MUSE ECC. For example, $M = 511$ provides SDDC guarantees and uses only nine bits of redundancy compared to the 12 bits needed by MUSE ECC. Second, due to aliasing and storage efficiency, Polymorphic ECC works with 8-bit and 16-bit symbols and standard 40-bit memory channels, an impossible task for MUSE ECC. Third, unlike MUSE ECC, where errors with remainder of zero would be misdetected, Polymorphic ECC relies on MAC for error detection, and thus can detect and correct such errors. Finally, the aliasing degree of a multiplier determines how many error candidates we have to try in the worst case to correct an error. Thus, by choosing a smaller multiplier value, i.e., a bigger aliasing degree, we make correction times longer but gain more space for the MAC, as we can see from Figure 7, which shows how with smaller multipliers, the aliasing degree and available space for MAC are increasing. The error-bars indicate, that some multipliers in the same bit-budget may have dramatically different aliasing degree. In Section VIII-A we show multiple variants of Polymorphic ECC differing in their MAC size as an example of this trade-off, while in the next section, we explain how Polymorphic ECC corrects errors.

C. Multi-Fault Model Error Correction

The error correction of Polymorphic ECC is based on the concepts of remainder polymorphism and iterative correction.

Remainder Polymorphism. The key feature of Polymorphic ECC is remainder polymorphism, i.e., its ability to interpret each remainder in different fault models. In practice, once the error is detected and its remainder is computed, we use the remainder's value to compute remainder-error mappings of each supported fault model to derive error candidates for correction. For example, the remainder of 86 can correspond to either a single bit flip in the second symbol or a four-bit

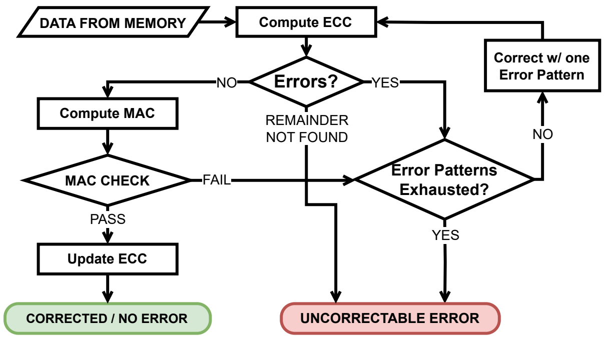


Fig. 8: The iterative process of error correction. The error is corrected when MAC check passes, otherwise an uncorrectable error is reported.

error in the first: $4096 \bmod 2005 = 86 \bmod 2005 = 86$ ($86 = 0101\ 0110_2$). As a result, we use single redundancy budget to cover many fault models that typically require more than one code, e.g., we can provide SSC, Double-error, and double Bounded-Fault correction in one code.

Iterative Error Correction. Figure 8 shows the process of error correction with Polymorphic ECC. First, the cacheline with its redundancy is read from memory, and remainders are computed for each codeword in the cacheline. If those remainders are zero, then a new MAC is computed from the data and verified against one stored in the cacheline. If they match, we declare no error and continue (we discuss Update ECC next). Otherwise, we start correcting by iterating through possible error candidates in all supported fault models until the error is corrected. Otherwise, an uncorrectable error is reported. After each correction attempt, we use the data from the corrected cacheline to compute a new MAC and compare it against one stored in the cacheline. Thus, if the MAC in the cacheline is corrupted by a correctable error, one of the attempts will correct it, and it will match one computed from the data. There is a small chance that one of the correction attempts will lead to MAC collision and silent data corruption, and we evaluate that chance in Section VIII-C.

Since MAC does not guarantee the integrity of ECC bits, there may be a situation where data and the MAC match, but ECC bits were miscorrected. The Update ECC step mitigates this issue: if the last iteration corrected errors in ECC bits, we compute new ECC bits from matching data and MAC, compare them, and report to the memory controller if a mismatch is detected. In our experiments, the rate of such errors was extremely low: ≈ 3.7 in $1{,}000{,}000$. Thus, since those errors are consistently detectable and correctable, we do not expect to see an impact on performance or reliability.

Let's see the following example of iterative correction with multiplier $M = 2005$ and 10-symbol codewords that store (from the right, Figure 6(b)) 11-bit ECC, a 5-bit MAC slice, and 64-bit data. Let's assume that upon reading the cacheline, one codeword's remainder is $R = 86$, indicating an error. With $M = 2005$, $R = 86$ has two error candidates: (86, *symbol-0*) and (16, *symbol-1*). Suppose the error is in *symbol-1*, representing a bit flip in the MAC slice: error integer *err_int* $= 16 << 8$ =0x1000. Following the flow in Figure 8, we use the first error candidate, i.e., (86, *symbol-0*), to correct the error: $C\_corr = C - Error(86, symbol\text{-}0)$ After the correction, we compute a new MAC value, and compare it against the one read from memory. Since the MAC read from memory still has one bit flipped, with high probability, we detect a MAC mismatch, discard the result of the prior correction, and try the next error candidate: $C\_corr = C - Error(16, symbol\text{-}1)$. We check all the remainders again, recompute the MAC, and compare it against the one read from memory. Unlike the first time, the second error candidate corrects the error in the MAC slice, and the cacheline passes the MAC verification step. At this point we recompute and compare ECC bits, signal to memory controller if a mismatch is detected, and report a Correctable Error.

TABLE III: Remainder Aliasing Degree vs. Multiplier Value

Multiplier	511	2005						
Aliasing Degree	10	1	2	3	4	5	6	7
Remainder Counts	510	368	520	528	328	130	22	2

Algorithm 1: Compute aliasing degrees.

```
input  : Multiplier M, number of symbols N, symbol width S
output: True if M defines a code and its aliasing degrees
/* Initialize empty REMS and ALIAS_DEG                    */
for symb_pos in 0..N-1 do
    /* generate err_ints for symbol at symb_pos */
    err_offset = symb_pos×$2^S$
    foreach err_int in 1..$2^S - 1$ do
        /* all symbol errors                              */
        add (err_int<<err_offset) to AllErrInts
    foreach err_int in AllErrInts do
        /* Remainders for both errors                     */
        $\mathtt{rem}_p$ = err_int  mod $M$
        $\mathtt{rem}_m$ = -err_int  mod $M$
        add $\mathtt{rem}_p$ and $\mathtt{rem}_m$ to set REMS
        update counts for $\mathtt{rem}_p$ and $\mathtt{rem}_m$ in ALIAS_DEG
    if len(REMS) $\neq 2 \times$ len(AllErrInts) then
        /* terminate, not enough remainders for
            correction within symbol                      */
        return (False, $\varnothing$)
    return (True, ALIAS_DEG)
```

Correction Latency. Since the number of error candidates per remainder is determined by the aliasing degree, the number of iterations it takes to correct the error depends on the multiplier value we chose. For example, Table III shows a histogram of the remainder aliasing degree for two Polymorphic ECC SDDC codes with multipliers 511 and 2005, having 56-bit and 40-bit MACs, respectively. For example, with $M = 511$, every remainder is aliased to 10 different errors, i.e., an error per symbol. With $M = 2005$, on the other hand, the majority of remainders have an aliasing degree of three, and only 2 are mapped to seven symbols. Thus, since the errors are corrected at cacheline granularity, in the worst-case scenario where each codeword in the cacheline is corrupted, with $M = 2005$, only

7^8 corrections are required instead of 10^8 with $M = 511$ – an improvement of 17×! In the typical case, however, the number of iterations is significantly lower – only 228. The main reason for this reduction is that only a tiny minority ($2/1898 = 0.1\%$) of errors have an aliasing degree of seven, and it is highly unlikely that errors in all the codewords in the cacheline would map to a remainder with an aliasing degree of seven (we evaluate error correction latency in Section VIII-C). Thus, with Polymorphic ECC, error correction speed can be traded off for stronger MACs at fine granularity, allowing deployment-specific customizations.

D. Remainder-Error Mapping

The simple solution to determine error candidates per remainder is to use lookup tables as in MUSE ECC. However, we realize that we can minimize storage overheads by deriving error value at runtime with the reverse relationship between the errors and remainders:

$$err\_int = (R \times Inv(2^L)) \bmod M \tag{2}$$

where err_int is the integer representation of error in a symbol that starts at bit offset L (line 2 in Algorithm 1), and $Inv(2^L)$ is the multiplicative inverse modulo M of the symbol at bit L, i.e., $(2^L \times Inv(2^L)) \bmod M = 1$. By construction, err_int has to fit into the symbol width. Otherwise, we know that the symbol that starts at bit L has no error with remainder R. For example, let's solve this equation for $symbol$-1 and $symbol$-2 and remainder $R = 86$ for $M = 2005$. With the $Inv(2^8) = 1026$ and $Inv(2^{16}) = 51$, we get $err\_int_1 = 16$ and $err\_int_2 = 376$. Since, err_int_2 does not fit into an 8-bit symbol, we discard it, while keeping err_int_1 as it is a valid error candidate for $symbol$-1. This way, for SDDC we do not need to store the mapping of remainders to errors, as we can derive those at runtime.

For the fault models with two corrupted symbols, e.g., double bounded fault, the naïve approach of finding two symbol errors that result in remainder R is computationally expensive because it requires to solve a system of equations. Thus, we store hints with locations of the errors and the error value for one of them. Then, we can easily solve the following equation to derive the error value for the first symbol:

$$err\_int_1 = (R - err\_int_2 \times Inv(2^{L_1})) \times Inv(2^{L_1}) \bmod M \tag{3}$$

We discuss the format of the hints in Section VI-B and evaluate storage overheads in Section VIII-D.

VI. Microarchitecture

A. System Integration

Figure 9(a) shows how Polymorphic ECC is integrated within memory write and read paths in the system. On the write path, first, the MAC is computed for each cacheline, then the MAC is sliced, and each slice is embedded into one codeword. Finally, an ECC is computed for both the MAC slice and the data, attached to the codeword, and it is written to memory. On the read path, the decoder (Figure 9(d)) for each of the codewords computes both the remainder and its error-candidate entry (Figure 9(b)-(c)). After the decoding, CW_AGGRTR gathers all the codewords and recovers the embedded MAC to compare it with one computed from the data. If MACs match we output the data to the CPU. Otherwise, we start iterative correction to correct errors in the cacheline and its MAC. We check the result by comparing the MAC extracted from the corrected cacheline (CORRCTD_EMBD_MAC) with one computed from corrected data (NEW_MAC). When MACs match, we use the MATCH signal to stop the correction. If MACs never match after all error candidates are tried, we declare an uncorrectable error by setting DUE to true.

B. Error-Candidates Generation

Figure 9(b) shows how P_ENTRY stores the information from the Error-Candidate Generator or ECG (Figure 9(c)) in the decoder (Figure 9(d)) about the remainder's error candidates. The ECG uses an ERR_INT_GEN unit per symbol, i.e., implements Eq. 2, or, for the double symbol fault models, each unit implements Eq. 3 by using hints, i.e., locations of faulty symbols and one error integer.

In general, P_ENTRY has a header and an aliasing degree number of sub-entries, where the header specifies the number of usable sub-entries as some remainders have no aliases. Each P_ENTRY can be used for either single or double-symbol fault models, and the main difference is in the information stored in each sub-entry. For example, for the single symbol model, each sub-entry stores the symbol's position and error integer, which the corrector uses to correct the error. For the double symbol models, each sub-entry stores the location of both faulty symbols and the err_int of the first symbol error (the second one is derived by Eq. 3). For example, for DDR5 SDDC with 8-bit symbols, each sub-entry is 13 bits and P_ENTRY itself is 81-bit long (header and six sub-entries).

C. Encoding and Decoding

The encoder (gray box in Figure 9(a)), decoder (Figure 9(d)), and the ECG (Figure 9(c)) use fast modulo circuits, which we implemented following the methodology from the original MUSE ECC paper [51]. Our decoder computes codeword's remainder and uses it to derive P_ENTRY with error candidates (or load and derive for double-symbol faults). Since the common case is no errors, the decoder passes the codeword to the output with minimal delay. In contrast, the P_ENTRY's sub-entries are pruned and reordered by the PRUNER & REORDERER circuit. This circuit outputs P_ENTRY but without sub-entries, which, if used for correction, will cause an overflow (or underflow in case of subtraction) in the symbol. For example, if the stored error integer is "-10" and the symbol value is "5", the result of correction will be "-5", an underflow. An underflow (or overflow) happens due to aliasing, and eliminating those entries speeds up error correction.

D. Iterative Error Correction

Figure 9(e) shows the Iterative Corrector's microarchitecture. The inputs are the STOP signal, the codewords, and their P_ENTRYs. The outputs are LAST_ITERATION to signal no

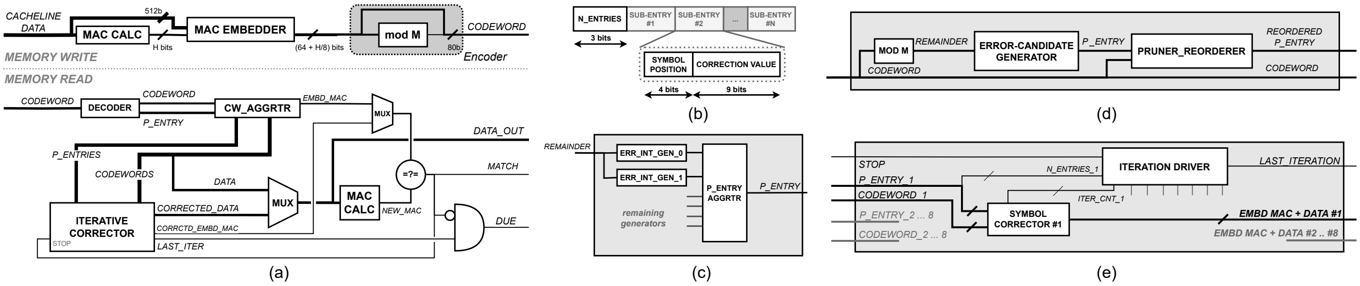


Fig. 9: (a) System integration of Polymorphic ECC. The upper half shows memory write path with MAC embedding and codeword encoding, while lower half shows memory read and error correction. (b) Format of the `P_ENTRY`: 3-bit header, and 13-bit sub-entries with fault position (four bits) and signed error integer (nine bits). (c) Error-Candidate Generator unit, where each `ERR_INT_GEN` implements Eq. 2 or Eq. 3 (table with hints is not shown). (d) Decoder consists of remainder calculation unit ($modM$), error-candidate generator (generates `P_ENTRY`s), and `PRUNER & REORDERER` to filter `P_ENTRY`s for subsequent error correction. (e) Iterative Corrector (for brevity only $codeword_1$ is shown).

Algorithm 2: Counting For Iterative Correction

```
input  : Counter Limits limits, Stop
output: Counters, Last_Iteration
Last_Iteration set to 0
while not Last_Iteration do
    increment counter[0]
    foreach counter[i], i in Counters do
        /* Start from lowest counter: i is 0      */
        if counter[i] == limits[i] then
            counter[i] set to 0
            if i is last then
                set Last_Iteration to 1
            else
                increment counter[i+1]
    yield Counters, Last_Iteration
```

more error candidates to try and data with MAC slices after the correction attempt. The STOP signal stops the iterative correction signal once MACs match. The ITER_DRVR unit uses the header fields of (already pruned and reordered) P_ENTRYs to determine the number of error candidates per codeword. In a nutshell, ITER_DRVR is a multidimensional counter where the header field of P_ENTRY limits each dimension, and in each clock cycle, only one counter value is incremented (see Algorithm 2 for pseudocode implementation). These counter values select error candidates per codeword for the correction.

Consider a toy example with two codewords, $C1$ and $C2$, whose P_ENTRYs have headers with values 3 and 2. During the correction, the ITER_DRVR will generate selector counters as $(1, 1)$, $(2, 1)$, $(3, 1)$, and $(3, 2)$. For example, with $(1, 1)$, both codewords will use their first error candidates. If, after the correction, MAC verification fails, ITER_DRVR will select $(1, 2)$, i.e., $C1$ will use its first error candidate, while $C2$ will be corrected with the second. For each correction attempt, we use the cacheline as it was read from memory. The correction procedure repeats until the corrector tries all covered fault models (LAST_ITERATION is raised) or MACs match (indicated by STOP). When LAST_ITERATION is raised, the DUE is set to report an uncorrectable error.

VII. Experimental Methodology

A. Estimation of the Out-of-Model Fault Rate

Unfortunately, there is little evidence of the actual Out-of-Model Fault rates in real systems today. Most recent studies (e.g., [8], [12], [67], [79]) use ECC and its logging infrastructure in CPUs to measure the rates of correctable and detectable errors, which by itself will distribute Out-of-Model errors as either correctable or detectable and also will skew the In-Model Fault rate [7]. To illustrate this point, let us consider a simple example with Hamming SEC-DED ECC. Say memory has 100 errors, with 80 single-bit, ten double-bit, and ten triple-bit errors. Suppose we use the ECC to classify those errors. In that case, by using miscorrection rates from Table II, we will measure 87 single-bit and 13 double-bit errors, completely ignoring the existence of triple-bit errors and overcounting the single-bit error rate by almost 9%. While some may argue that SEC-DED ECC is not designed to handle triple-bit errors, using ECC logging undercounts Out-of-Model Fault rates due to miscorrection as evidenced by the following equations:

$$\begin{aligned} N_{single\ error} + 0.7 \times N_{triple\ error} &= 90 \\ 0.3 \times N_{triple\ error} + N_{double\ error} &= 10 \end{aligned} \tag{4}$$

As a result, since we have only two equations for three variables, we cannot determine the real fault rate as there is an infinite number of solutions that satisfy these equations. Thus, in our evaluation, when comparing Polymorphic ECC with other memory reliability schemes, we will compare SDC rates per fault model for each scheme. Those result can be weigthed to estimate code performance for specific fault distributions and system's operating settings to estimate overall reliability guarantees with Polymorphic ECC.

For the comparison, we evaluated two classes of codes. First is the SDDC RS code, miming those in commercial products [21], [46], and second is Unity ECC [41], and Bamboo ECC [42], which cover more fault models than SDDC RS. Unity ECC extends SDDC RS code to cover double-bit errors, while Bamboo ECC uses long (half cacheline and full cacheline with 8-bit and 16-bit symbols, respectively) RS codewords with symbols aligned to DRAM pins. To provide ChipKill guarantees, Bamboo ECC corrects four symbol errors, i.e., symbol per pin of a failing DRAM device.

B. Fault Injection Methodology

To emulate miscorrected errors that propagated to the memory state of the application, we use CRIU (Checkpoint and Restore In Userspace) – a Linux utility that checkpoints the program's memory state to disk and restores it later for 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 profile the programs to ensure that 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 checkpoint command by t_{inj}. After the checkpoint, we uniformly pick a random injection address A_{inj} in its address space. This way, the injection time and address are randomly sampled from the program's memory state. We use the same checkpoint state, t_{inj}, and A_{inj}, for both encrypted and non-encrypted memory models to guarantee that both experiments 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 profiled the RS code to obtain errors that represent DRAM failures leading to miscorrection. For each injection, we randomly select one of these errors, e_{inj}, and use it to corrupt the checkpoint at address A_{inj} in both memory models. For the plaintext memory model, we use the profiled errors as-is. For the encrypted memory model, we create encryption-amplified errors: we encrypt the cacheline at address A_{inj} with AES, corrupt the ciphertext with e_{inj}, and then decrypt it. As a result, the decrypted cacheline is corrupted with an encryption-amplified error pattern. We write this cacheline back into the memory image and resume execution. This process simulates how the application's memory state would be corrupted if it were running on a system with memory encryption and experienced a DRAM failure miscorrected by RS ECC.

Outcome Classification In line with prior work [39], we assume that if the program's execution is longer than 3× of its remaining error-free execution time, it enters a bad state, and we terminate it. We categorize injection outcomes into the following categories:

- Hang – 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 as in fault-free case.

C. Experimental Setup

Fault Injection. We use the methodology by Leveugle et al. in [47] 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 reduce execution times, we use train inputs for the benchmarks.

Inference We relied on ONNX Runtime C++ APIs and the ONNX neural network model to study inference workloads. We used the image classification Mobilenet v2 ONNX model for the ML experiments with the ImageNet 2012 validation dataset. We randomly sampled 2,500 images from the 50,000 available to keep runtimes reasonable. For the FHE experiments, we relied on the HE-Man-Concrete Library [57], [77]. Given the long execution times of the FHE inference, we used 100 images from the MNIST database of handwritten digits and the classification ONNX model from the HE-Man-Concrete GitHub repository.

Performance Analysis. To measure the impact of Polymorphic ECC's hardware and its latency, we modified gem5 [9] simulator, so that writes are delayed, emulating the addition of encoders and MAC unit. Since Polymorphic ECC is a systematic code, there is no delay on the read path. We used SPEC'17 benchmarking suite, updated to the v1.1.9 [10], compiled with GCC 9.4.0 and optimization level -O3 under Ubuntu 20.04. We configure gem5 with 3.4GHz CPU, 64kB K1, 256kB L2 for each core, 8MB of L3, and 32GB of RAM. To get a conservative estimate on the performance impact, we used TimingCPU, which models memory accesses in detail, while executing instructions in one clock cycle. We executed the benchmarks for 400M instructions with reference inputs.

VIII. Evaluation

In this section we answer the following research questions:

A) How Polymorphic ECC can be configured?
B) What fault models Polymorphic ECC covers?
C) How fast Polymorphic ECC corrects errors?
D) What are the hardware and performance overheads of Polymorphic ECC?
E) How does Polymorphic ECC corrects Rowhammer-induced errors?

A. Polymorphic ECC Configurations for DDR5

Unlike MUSE ECC, which is limited to 4-bit symbols and wide memory channels, Polymorphic ECC supports 8-bit and 16-bit symbols, which alleviate the need for wide memory interfaces and reduce the overall power of the memory subsystem. Table IV shows various DDR5-compliant variants of Polymorphic ECC with their supported fault models. We discuss the main differences between the configurations due the symbol size and the multiplier value.

Symbol Size. Symbol size affects code configuration in two ways: wider symbols increase codeword size, and require larger multipliers. For example, when we increase symbols size from 8-bit to 16-bit, the codeword length increases from 80 to 160 bits, taking twice as many beats to read or write from memory. In addition, since wider symbols have more

TABLE IV: Aliasing Degrees or Fault Models

Symbol Size	M	Fault Model	Aliasing Degrees Max	AVG±STD	MAC bits
16b	131049	SSC	11	10 ± 0.04	60
		DEC	3	1.14 ± 0.38	
8b	511	SSC	10	10 ± 0	56
	1021	SSC	10	5 ± 1.58	48
		DEC[†]	18	11.27 ± 2.45	
	2005	SSC	7	2.69 ± 1.23	40
		DEC	12	5.75 ± 2.05	
		BF+BF	101	78.81 ± 6.50	
		ChipKill+1	436	355 ± 14.50	

[†]: **bold** shows a new fault model enabled by a larger multiplier value.

possible errors, they need more remainders and thus larger multiplier values. For example, with 16-bit symbols, we need a multiplier of 131,049 compared to 1021 for 8-bit symbols to support the same fault models.

Multiplier Values. The multiplier value directly affects what fault models are supported by the code, average error correction latency, and length of the MAC. First, larger multipliers have enough remainders for fault models with many errors, e.g., ChipKill+1 has 183,600 errors compared to 2,550 of ChipKill. Second, for the same symbol size, larger multiplier value has lower aliasing degree, which leads to fewer iterations to correct an error. For example, for the SSC fault model, the average number of iterations decreases from 570,421 for $M = 511$ to 6620 for $M = 1021$ and 228 for $M = 2005$ (Section VIII-C for more details). Thus, for the same redundancy budget it is preferable to use the largest multiplier that satisfies all the constraints. In addition, some multipliers do not map zero remainder to errors, which ensures that in the common case of no errors no time is spent to correct an error that may not exist. For example, both $M = 2041$ and $M = 2005$ support DEC and BF+BF fault models. However, with $M = 2041$ DEC and BF+BF have 2 and 102 errors mapped to zero remainder, which is not the case with 2005.

However, some fault models have errors that map to zero remainder, e.g., ChipKill+1 with 218 errors mapped to zero remainder. To minimize the impact on performance of those models in the common case, we propose to check against this model last, and do so in two phases. First, we ignore the errors that mapped to zero remainder. If the error persists after phase one, we try again without excluding error that map to the remainder of zero. This way, we exclude a very small number of errors (218 out of 183,600), while minimizing the impact on the common case.

Overall, Polymorphic ECC's family of codes is flexible, works with several symbol sizes, and may be adapted to other memory technologies like HBM3 [35]. However, since the interfaces and fault models of HBM3 differ from DDR5, a detailed study is required to assess necessary tradeoffs.

B. Fault Coverage with Polymorphic ECC

Table V compares the error correction performance of Polymorphic ECC and other codes across various fault models: ChipKill – corrupts every codeword at the same symbol with random error, SSC – random symbol error in every codeword, DEC – two random single-bit errors in a codeword, BF+BF – aligned double bounded fault per codeword, and ChipKill+1 – ChipKill with a failed pin on a second DRAM chip. We see fault coverage of each code, i.e., In-Model vs. Out-of-Model, where the reported data is for Out-of-Model faults in cells with a gray background. Polymorphic ECC supports all fault models as In-Model, while Unity ECC is the second supporting ChipKill, SSC, and DEC. Bamboo ECC covers only the ChipKill model due to its pin-aligned symbols. Thus, Unity ECC and RS cover the SSC model, while Bamboo ECC does not because in SSC, the codewords may have errors originating from different chips, which will corrupt more than four pin-aligned symbols of Bamboo ECC.

Based on the fault model coverage, we see that Polymorphic ECC is the only code that corrects all errors. Other codes correct only a subset of the fault models, e.g., Unity ECC corrects ChipKill, SSC, and DEC, while Bamboo ECC corrects only ChipKill errors. However, due to its iterative nature, there is a non-zero chance that Polymorphic ECC will have MAC collision in one of the iterations, leading to an SDC. We analyze SDC rates in detail in Section VIII-C. However, unlike other codes, Polymorphic ECC not only covers more errors but also provides in-lined MACs for security, which is especially important today with an increasing focus on security and confidential computing [4], [11], [16], [25], [40], [52].

C. Analysis of Iterative Error Correction

Here, we study how long it takes to correct an error and what is the chance of miscorrection. To do so, we created 10^5 random cachelines that were corrupted according to the respective fault model. For each of the fault models, we conservatively assume that every codeword has an error, e.g., for the SSC fault model, every codeword in the cacheline has one symbol-error, corresponding to an average bit error rate of $\approx 5 \times 10^{-2}$. The errors were fixed iteratively as described in Section V-C, and summarized the results in Table V.

Latency. As shown in Figure 8, the iterative correction latency is due to error correction attempts and consequent recomputations of the MACs. Thus, we can separate the latency into fixed and variable parts. The fixed cost, T_{fix}, is due to the codeword decoding, generation, pruning, and reordering of error candidates: $T_{fix} = T_{dec} + T_{pruning}$. The variable part, T_{var}, is due to the error candidate selection, correction, and MAC verification repeated N times: $T_{var} = N \times (T_{\text{ITER\_DRVR}} + T_{ECG} + T_{MAC}) = N \times T_{var,0}$. Overall, for the N-iteration correction, the latency is $T(N) = T_{fix} + N \times T_{var,0}$. Using hardware implementation results from Section VIII-D, the total latency is $T_{corr} = 3.98 + 5.36 \times N\ ns$. For example, ChipKill has the shortest correction time, requiring only one iteration, or 9.34 ns, shorter than the reported correction times of Intel's CPUs [14]. Polymorphic ECC's correction latency is also shorter than that of other MAC-based ECC schemes. For example, CSI:Rowhammer has an average correction time of five hours for an 8-bit error, which is due to the use of parity to limit the search space,

TABLE V: Fault coverage and error correction performance of Polymorphic ECC and other codes.

Symbol Size	Fault Model$^{\alpha}$	Polymorphic ECC			Reed-Solomon		Unity ECC [41]		Bamboo ECC [42]	
		# Iters, Avg±Std	SDC	DUE$^{\beta}$	SDC	DUE	SDC	DUE	SDC	DUE
16b	`ChipKill`	1.30 ± 0.56	1.13×10^{-18}	N/A	0	0	0	0	0	0
	`SSC`	468 ± 407	4.1×10^{-16}		0	0	0	0	0	0.999
	`DEC`	1.64 ± 1.25	1.43×10^{-18}		0.2	0.98	0	0	0	0.999
8b	`ChipKill`	1	0	N/A	0	0	0	0	0	0
	`SSC`	228 ± 493	2.1×10^{-10}		0	0	0	0	1.7×10^{-5}	0.998
	`DEC`	$554{,}132 \pm 1{,}073{,}304$	5.0×10^{-7}		0.024	0.976	0	0	2.5×10^{-5}	0.999
	`BF+BF`	65 ± 108	5.89×10^{-11}		0.03	0.97	0.065	0.871	2.2×10^{-5}	0.999
	`ChipKill+1`	$4{,}464 \pm 7{,}516$	4.1×10^{-9}		0.03	0.97	0.062	0.813	2.4×10^{-5}	0.999
8b	Rowhammer Patterns	2.52 ± 5.80	0	1.7×10^{-4}	4×10^{-4}	11.3×10^{-3}	0	2.5×10^{-4}	0	0

α: 80-bit codewords with symbol folding [21]. β: All errors are correctable. ▨ Out-of-Model Faults for respective codes.

which is prone to hide errors and thus needs more steps to identify those, e.g., two-bit error within the same symbol [38].

While variable error correction latency is acceptable [14], [75], it has to be small enough not to convert a benign error into a denial-of-service event, e.g., Intel SGX processor lockdown [24], [32], making workload migration due to a memory failure, a common feature of deployments at scale [17], [28], challenging. With Polymorphic ECC, correction latency can be bounded in multiple ways. First, fault models with long correction latencies may be disabled without significantly impacting error coverage, e.g., `DEC`, since `BF+BF` and `ChipKill+1` cover most of the double-bit errors. In this case, the longest correction latency would be $3.98 + 5.36 \times 4,464 = 23.93\ \mu s$. An alternative approach is to limit the number of error correction iterations to N_{max} while supporting all possible fault models, declaring DUE if MACs do not match after N_{max} iterations. For example, for the 8-bit symbol code, 99.73% of the errors (3-sigma) are corrected within $N_{max} \approx$ 3,000,000, leading to $T = 3.98 + 5.36 \times$ 3,000,000 $\approx$ 16.1 ms, which is comparable to correction latencies of some Intel CPUs [14]. As a result, Polymorphic ECC can be tuned to various deployment settings while providing better error coverage than comparable codes.

Reliability. As we see from the table, Bamboo ECC has the lowest SDC and highest DUE rates among RS-based codes for Out-of-Model faults. This outcome is expected because Bamboo ECC uses long codewords and large symbols, a common technique to minimize SDC rates [36], resulting in SDCs being converted into DUEs. For those fault models, the 8-bit-symbol Polymorphic ECC has at least 100× lower SDC rates than Bamboo ECC, while with 16-bit symbols, Polymorphic ECC comes in second with extremely low SDC rates of 1.43×10^{-18}. Overall, except for the `ChipKill` fault model, Polymorphic ECC is expected to have non-zero SDC rates due to iterative correction.

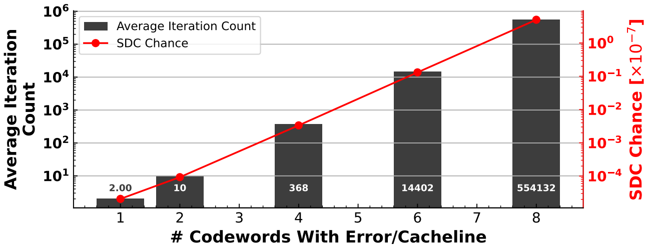


Fig. 10: The average number of iterations it takes to correct a `DEC` error (gray bars) and SDC rate (red line) vs. number of corrupted codewords per cacheline (as a proxy to BER). Y axis is in log scale.

However, in actual deployments, those rates may be even lower. First, the SDC rates with Polymorphic ECC heavily depend on the operational environment and bit error rate. Figure 10 shows how the average error correction iteration count behaves with an increase in the number of corrupted codewords, a proxy metric we use for bit error rate. From the figure, we see that both the number of iterations (gray bars) and the SDC rates (red line) grow exponentially with the increase in error rates, i.e., the number of corrupted codewords per cacheline (Y axis is in log scale). From the shown SDC trend, we can estimate that with more realistic error rates, SDC rates of Polymorphic ECC would be much lower than presented in Table V, which assumes the conservative scenario of eight corrupted codewords per cacheline.

Second, the ECCs available in commercial CPUs tend to be deployed conservatively [21], [41], prioritizing detection over correction, essentially choosing `ChipKill` over `SSC`. Moreover, to minimize the chance for Out-of-Model errors, datacenter operators proactively replace DIMMs [18], [48] after a specific number of correctable errors, as few as 100, as reported in [53]. In those settings, the chance of SDC with Polymorphic ECC after 100 errors is $1 - (1 - p_{SDC})^{100} = 2.1 \times 10^{-8}$ (or 4.4×10^{-16} for 16-bit symbol code), where p_{SDC} is the probability of SDC for one `SSC` error (see Table V). Thus, Polymorphic ECC is a practical alternative to the RS-based codes as it covers more Polymorphic ECC errors and provides data integrity with up to 60-bit MAC at the cost of very low SDC rates.

In addition, workloads deployed at scale already heavily use software-based checksums for in-memory data for SDC detection [6], indicating that hardware-software co-design processes are not new, and a promising avenue for future work is to explore the software co-design with Polymorphic ECC to lower SDC rates even further. As a result, Polymorphic ECC offers superior error coverage than the RS-based codes and

TABLE VI: Hardware Implementation Results, $M = 2005$

Circuit	Latency, ns	Area, μm^2	Power, W
Encoder/Decoder	2.52	25,565	1.801
Qarma [5]	1.636	22,549	2.95
ITER_DRVR	0.96	548	0.001
PRUNER & REORDERER	1.46	3,857	0.003
ECG (10 symbols)	2.76	46,319	2.156
ERR_INT_GEN (Eq. 2)	2.2	5,906	0.189
Symbol Size	**Hint Storage, kB**		
8b	DEC: 17	BF+BF: 259	ChipKill+1: 892
16b	DEC: 143	-	-

can further reduce the memory share of datacenter ownership costs, which are already similar to those of CPUs [50], e.g., fewer DIMM replacements and related workload migrations, lower SDC rates due to Out-of-Model faults, etc.

D. Hardware and Performance Overheads

Hardware Overheads. We implemented our circuits in Verilog (Qarma's implementation was available online[1]), and synthesized with OpenROAD open-source VLSI toolchain [2], [3] with integrated NangateOpenCell 45nm open-source standard cell library [66]. The results are summarized in Table VI. As expected, the longest latencies are Qarma and units that use modulo computation circuits, i.e., ECG (uses ten ERR_INT_GEN in parallel), encoders, and decoders. The iterative corrector and pruner with reordering have smaller latencies and area overheads. However, similarly to MUSE ECC [51], compared to other codes, Polymorphic ECC uses significantly more area and longer latency mainly due to its reliance on integer multipliers rather than CMOS-friendly XOR operations. For example, the encoder of Polymorphic ECC has a delay of eight full adders and one carry-look-ahead adder. In contrast, Unity ECC has only 7 XOR gates, which is significantly faster. Given the die sizes of modern CPUs, extra hardware of the Polymorphic ECC is not expected to cause a significant area or power increase. For example, ARM devices already implement Qarma in hardware, while Intel TDX implements a KECCAK-512-based MAC unit [76].

Storage of hints. However, some fault models require solving Eq. 3, which, as formulated, needs hints, i.e., symbol locations and one error integer. The last two rows in Table VI show the storage overheads of the hints. It is not uncommon to use syndrome storage for error correction; for example, some AMD CPUs use 10kB syndrome tables [1]. We leave alternative ways to derive the hints at runtime, e.g., feeding all possible combinations of fault locations to ERR_INT_GEN units to find error integers that solve Eq. 3 to future work.

Performance Overheads. We used 4.2 *ns* on the write path to account for the codeword encoding and computation of the MAC (see Table VI). We assume TDX-like baseline, where MAC decoding is always enabled. Figure 11 shows the results of the execution, normalized to a baseline without delayed writes. On average, performance penalty is $\approx 1\%$, while some workloads showing slowdown of $\approx 3\%$ (e.g., deepsjeng and roms). This is expected as writes are rarely on critical path, and extra latency is small compared to the modern systems' memory latencies, which are in 100s of ns range [13].

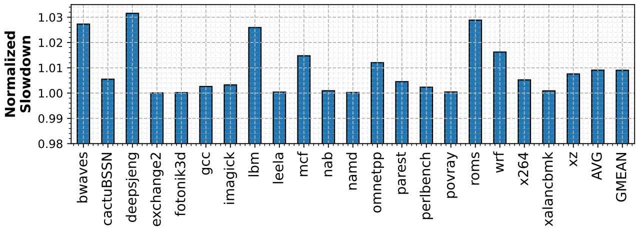


Fig. 11: Normalized slowdown due to the encoder and MAC unit of Polymorphic ECC.

E. Rowhammer Case Study

In the last row of Table V are the evaluation results of all the codes on 94,892 cacheline-long rowhammer patterns kindly provided to us by Venugopalan *et al.* [73]. All the codes corrected most of the patterns, as only 1113 out of 94,892 have more than one corrupted symbol per codeword: 1091 with double-bit errors and 24 with three-bit errors. The Reed-Solomon code performed the worst, with SDC and DUE rates of 4×10^{-4} and 11.3×10^{-3}, respectively, while Bamboo ECC outperformed all other codes as it can correct up to four symbols, while the most severe patterns had three bit-flips. Polymorphic ECC came in second with 16 DUEs (all due to random triple-bit errors), correcting eight more three-bit patterns that Unity ECC could not (24 DUEs). The main reason for the better performance of Polymorphic ECC is correctable bounded fault model (BF+BF), which Unity ECC does not support, as some of the three-bit patterns were aligned with double bounded faults. From the latency perspective, on average, Polymorphic ECC corrected a rowhammer-induced error in 2.68 iterations (with std_dev=7.49), which is much faster than the conservative estimates for BER of 0.05 in Table V. If we assume that benign and intentional rowhammer errors we evaluated here are similar, Polymorphic ECC may significantly raise the cost of a rowhammer-based attack and correct many benign rowhammers, while also offering data integrity with MACs in the common case.

IX. Related Work

Historically, ECC used for main memory is based on codes that use mod 2 arithmetic, as those were simpler to build and thus more reliable [30], [49]. With DRAM becoming more popular and the introduction of the internet in the 90s, memory reliability became a marketable feature as we started to use computers differently [15]: e-commerce, client-server model, etc. These changes required stronger ECC than SEC-DED. Since then, the de facto standard error correction offered by the vendors today is Single DRAM Device Correct (SDDC) ECC [21], [27], [69]. Higher guarantees would require trading off memory parallelism [27], [69] or proprietary memory modules [27]. Unlike those schemes designed for a single fault model,

[1] https://github.com/ammrat13/spring2022-cs3220-aos

Polymorphic ECC covers multiple fault models using standard redundancy budgets at the cost of iterative error correction.

Academic SDDC schemes aim to exceed guarantees of SDDC via different approaches, which include co-design with DDR5's on-die ECC (e.g., DUO ECC [23], XED [55], OBET [56]), use clever codeword organizations (e.g., Bamboo ECC [42], Multi ECC [36], LOT ECC [72], and others), or data duplication and disaggregation (e.g., Dve [59]). Others co-design ECC for selective data protection (Context-Aware Resiliency [63]), or security (MUSE ECC [51], AFT-ECC [68]). IVEC [31], SYNERGY [62], ITSEP [70], and more recent SafeGuard [20] and CSI:Rowhammer [38], use MACs for detection, and error correction through search. However, unlike Polymorphic ECC, those solutions either restrict the search space and, thus, error coverage or require interaction with the OS and have correction times for multi-bit errors in the order of hours (CSI:Rowhammer). In contrast, Polymorphic ECC does it faster and needs no software support.

Another avenue for reliability-security co-design is to combine tagging for memory safety, data integrity and confidentiality, and resiliency to memory errors. Two recent solutions, Voodoo [45] and HashTag [44], both use MACs for error detection while also allowing some storage to be used for memory safety tags. The core component of Voodoo is MAGIC [43], a novel MAC design that combines encryption, authentication, and error correction. With MAGIC, Voodoo can embed up to 36-bit tags per cacheline while correcting a subset of `ChipKill` errors. Similarly, HashTag trades off reliability by reusing ECC storage for tags and MAC. As a result, the errors are corrected through search guided by parity bits, resulting in long correction times. Both of those techniques offer weaker reliability guarantees than Polymorphic ECC as they convert Out-of-Model errors to DUEs. Moreover, since Polymorphic ECC is MAC-agnostic, it can support memory tagging embedding in a MAGIC-like fashion; we leave the study of such integration to future work.

ARCC [37] additively increases strength of ECC only for memory regions that show more errors during memory scrubbing. While ARCC uses standard ChipKill, Polymorphic ECC can make the idea even more advantageous, as one can use single ECC schemes for all the memory, while choosing a better fault model when error rates rise. Similarly, ArchShield [54] can assign stronger fault model for vulnerable words instead of storing them in separate storage.

The ECC schemes for emerging technologies, like nvm-based persistent main memory, must provide higher reliability guarantees than DRAM-specific ECC because nvm-based main memory allows data persistence during system reboot. The power-off phase preserves the data and accumulates errors, necessitating ECCs more errors than just ChipKill, e.g., [78] with multiple tiers of codes. Polymorphic ECC's support of multiple fault models may further improve those systems' reliability. Moreover, Soteria [80], a reliability-security co-design of nvm-based main memory, may benefit from Polymorphic ECC's integrated MACs, to simplify its design.

Perhaps the closest in spirit scheme to Polymorphic ECC is Unity ECC [41], which uses unused syndromes in the code for double errors. However, Unity ECC's error coverage is limited by the number of unused syndromes and leaves no space for MACs, resulting in a less secure system than one with Polymorphic ECC.

X. Conclusion

This paper showed how Out-of-Model Faults affect systems' availability and reliability with encrypted data. To mitigate the effects of those faults, we presented Polymorphic ECC – a novel ECC scheme with *redundancy polymorphism*, which allows the same redundancy value to be used for multiple fault models, resulting in more efficient storage utilization. We showed that with Polymorphic ECC's novel construction, the search space for errors is much smaller than when parity-based schemes are used due to fine-grained control of the redundancy size while preserving symbol alignment to DRAMs. As a result, traditional fault models, e.g., `ChipKill`, are corrected in one iteration, compared to the parity-guided search that needs orders of magnitude more trials. This feature is the core strength of residue codes that differentiates it from schemes based on Galois Fields. As a result, Polymorphic ECC corrects more errors faster, making it highly practical.

In addition, we showed that Polymorphic ECC is highly flexible, allowing embedding any MAC, which can be at least 40-bit long, thus providing better security and data integrity than commercial systems while detecting almost any error. To showcase its utility, we showed how Polymorphic ECC corrects a series of a real-world rowhammer-induced errors, which was possible due to its extended fault model coverage.

Looking forward, the expansive error coverage with Polymorphic ECC can enable precise studies of DRAM failures in the field, such as recent efforts proposed for standardization by the Open Compute Project (OCP) for Memory Fault Management Infrastructure (FMI) [60] across the vendors. With Polymorphic ECC and OCP's FMI, detected errors could be classified with more precision than with the traditional codes, which convert Out-Of-Model Faults into SDCs or DUEs, hiding their true nature and scope. Overall, Polymorphic ECC represents an exciting and novel advance for co-designing secure and reliable systems with low overhead.

Acknowledgment

We thank Hari Venugopalan, Kaustav Goswami, and Professor Jason Lowe-Power for providing us with rowhammer error patterns for the experiments. We also thank Google for the unrestricted gift to support this research. Generative AI tools were utilized to generate figures and parts of the source code in the artifact. Simha Sethumadhavan has a significant financial interest in Chip Scan Inc.

Appendix

A. Abstract

This artifact contains the following three components

A) **ECC Miscorrection Profiler**: code to reproduce Table II.

B) **SDC Profiler**: code to reproduce the results in Table IV, Table V, and Figure 7.

C) **VLSI implementation**: Verilog implementation of Polymorphic ECC to reproduce Table VI.

Artifacts are packaged with Docker, so they are easy to set up and run.

B. *Artifact check-list (meta-information)*

- **Algorithm:** Source code implementing Algorithm 1 and 2
- **Compilation:** performed automatically within the docker container.
- **Run-time environment:** Docker container.
- **Hardware:** x86_64-based system.
- **Execution:** No requirements.
- **Metrics:** printed out tables, figures as pdf files.
- **Output:** A text file per experiment with search configuration supplied via command line, i.e., codeword and symbol lengths in bits, redundancy budget in bits, etc., and a table with data. Figures as pdf files.
- **How much disk space is required (approximately)?:** about 30-40 GB.
- **How much time is needed to prepare workflow (approximately)?:** few minutes for all artifacts.
- **How much time is needed to complete experiments (approximately)?:** Few hours for artifact 1, 5-7 days for Artifact 2, and about 20 minutes for Artifact 3. One specific configuration of Artifact 2 is the bottleneck, others can be done in about a day (depending on the CPU count).
- **Publicly available?:** Yes, archived via Zenodo.
- **Code licenses (if publicly available)?:** APACHE 2.0
- **Workflow automation framework used?:** Docker containers.
- **Archived (provide DOI)?:** https://doi.org/10.5281/zenodo.13786095

C. *Description*

1) How to access: The artifact is licensed with Apache 2.0 license and can be downloaded from https://doi.org/10.5281/zenodo.13786095. The artifact contains the source code, installation instructions, and steps to run the experiments.

2) Hardware dependencies: The artifact was developed and tested on x86_64-based system with Intel Core i7-8700 CPU.

3) Software dependencies: The artifact requires Docker runtime.

D. *Installation*

Download and install docker daemon via https://docs.docker.com/get-docker/. Each directory of the artifact has Makefile that builds the container and produces the results, i.e., tables and figures.

E. *Experiment workflow*

ECC Miscorrection Profiling In `profiling` directory, reproduce Table II by running `make missdata`. The results are saved to a file named `Table-2.txt`.

SDC Profiling In `profiling` directory, reproduce Table IV and Table V by running `make sdcdata`. The results are saved to files named `Table-4.txt`, `Table-6.txt`.

Aliasing Degree Trade-off In directory `tradeoff-figure`, reproduce Figure 7 by running `make figure`. When profiling is done, the figure will be in file `Figure.pdf`.

VLSI implementation The experiments are performed with `make hw`. After all place-and-route runs are finished, a table with the results will be printed to the console and saved to a file named `Table-5.txt`.

F. *Evaluation and expected results*

At the end of the simulations, the content of all the files should be close to those in the paper. A small variance is expected as all the profilers rely on random number generators.

G. *Notes*

SDC Profiling. The SDC profiler is moderately fast except for the `ChipKill+1` and `DEC` models. Those take about a week on a 96-core CPU for one million test cases, and can run significantly faster while providing similar results with fewer test cases.

H. *Methodology*

Submission, reviewing and badging methodology:

- https://www.acm.org/publications/policies/artifact-review-and-badging-current
- https://cTuning.org/ae

References

[1] *BIOS and Kernel developer's guide (BKDG) for AMD family 15h models 00h–0Fh processors*, Advanced Micro Devices, Inc., January 2013.

[2] T. Ajayi, D. Blaauw, T.-B. Chan, C.-K. Cheng, V. Chhabria, D. K. Choo, M. Coltella, R. G. Dreslinski, M. Fogaça, S. M. Hashemi, A. A. Ibrahim, A. B. Kahng, M. Kim, J. Li, Z. Liang, U. Mallappa, P. I. Pénzes, G. Pradipta, S. Reda, A. Rovinski, K. Samadi, S. S. Sapatnekar, L. K. Saul, C. Sechen, V. Srinivas, W. Swartz, D. Sylvester, D. Urquhart, L. Wang, M. Woo, and B. Xu, "OpenROAD: Toward a Self-Driving, Open-Source Digital Layout Implementation Tool Chain," in *Proceedings of Government Microcircuit Applications and Critical Technology Conference*, 2019, pp. 1105–1110.

[3] T. Ajayi, V. A. Chhabria, M. Fogaça, S. Hashemi, A. Hosny, A. B. Kahng, M. Kim, J. Lee, U. Mallappa, M. Neseem *et al.*, "Toward an open-source digital flow: First learnings from the OpenROAD project," in *Proceedings of the 56th Annual Design Automation Conference 2019*, 2019, pp. 1–4.

[4] *ARM Architecture Reference Manual Supplement for ARMv9-A architecture profile.*, ARM Limited, 2021, Rev. A.d.

[5] R. Avanzi, "The QARMA Block Cipher Family. Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes," *IACR Transactions on Symmetric Cryptology*, vol. 2017, pp. 4–44, 2017.

[6] D. F. Bacon, "Detection and prevention of silent data corruption in an exabyte-scale database system," in *The 18th IEEE Workshop on Silicon Errors in Logic-System Effects (SELSE)*, 2022.

[7] L. Bautista-Gomez, F. Zyulkyarov, O. Unsal, and S. McIntosh-Smith, "Unprotected computing: A large-scale study of dram raw error rate on a supercomputer," in *SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 645–655.

[8] M. V. Beigi, Y. Cao, S. Gurumurthi, C. Recchia, A. Walton, and V. Sridharan, "A Systematic Study of DDR4 DRAM Faults in the Field," in *2023 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2023, pp. 991–1002.

[9] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[10] J. Bucek, K.-D. Lange, and J. v. Kistowski, "SPEC CPU2017: Next-generation compute benchmark," in *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, Berlin, Germany, 2018, pp. 41–42.

[11] 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," *ACM Computing Surveys*, vol. 56, no. 9, apr 2024.

[12] Z. Cheng, S. Han, P. P. C. Lee, X. Li, J. Liu, and Z. Li, "An In-Depth Correlative Study Between DRAM Errors and Server Failures in Production Data Centers," in *2022 41st International Symposium on Reliable Distributed Systems (SRDS)*, 2022, pp. 262–272.

[13] A. Cho, A. Saxena, M. Qureshi, and A. Daglis, "A Case for CXL-Centric Server Processors," 2023.

[14] 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.

[15] T. J. Dell, "A white paper on the benefits of ChipKill-correct ECC for PC server main memory," IBM Microelectronics division, Tech. Rep., 11 1997.

[16] G. Dhanuskodi, S. Guha, V. Krishnan, A. Manjunatha, M. O'Connor, R. Nertney, and P. Rogers, "Creating the First Confidential GPUs: The team at NVIDIA brings confidentiality and integrity to user code and data for accelerated computing." *Queue*, vol. 21, no. 4, pp. 68–93, sep 2023.

[17] H. D. Dixit, L. Boyle, G. Vunnam, S. Pendharkar, M. Beadon, and S. Sankar, "Detecting silent data corruptions in the wild," 2022.

[18] X. Du, C. Li, S. Zhou, M. Ye, and J. Li, "Predicting Uncorrectable Memory Errors for Proactive Replacement: An Empirical Study on Large-Scale Field Data," in *2020 16th European Dependable Computing Conference (EDCC)*, 2020, pp. 41–46.

[19] M. Dworkin, E. Barker, J. Nechvatal, J. Foti, L. Bassham, E. Roback, and J. Dray, "Advanced Encryption Standard (AES)," 2001.

[20] A. Fakhrzadehgan, Y. N. Patt, P. J. Nair, and M. K. Qureshi, "SafeGuard: Reducing the Security Risk from Row-Hammer via Low-Cost Integrity Protection," in *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, 2022, pp. 373–386.

[21] J. Fruehe, "AMD EPYC brings new RAS capability," Moor Insights and Strategy, Tech. Rep., June 2017, white paper.

[22] R. Gilad-Bachrach, N. Dowlin, K. Laine, K. Lauter, M. Naehrig, and J. Wernsing, "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy," in *Proceedings of The 33rd International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. F. Balcan and K. Q. Weinberger, Eds., vol. 48. New York, New York, USA: PMLR, 20–22 Jun 2016, pp. 201–210.

[23] S. Gong, J. Kim, S. Lym, M. Sullivan, H. David, and M. Erez, "DUO: Exposing On-Chip Redundancy to Rank-Level ECC for High Reliability," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Vienna, Austria, 2018, pp. 683–695.

[24] D. Gruss, M. Lipp, M. Schwarz, D. Genkin, J. Juffinger, S. O'Connell, W. Schoechl, and Y. Yarom, "Another Flip in the Wall of Rowhammer Defenses," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018, pp. 245–261.

[25] M. Guevara, "Expanding our Fully Homomorphic Encryption Offering," https://developers.googleblog.com/2023/08/expanding-our-fully-homomorphic-encryption-offering.html, August 2023, accessed: 2023-11-28.

[26] D. S. Henderson, "Residue class error checking codes," in *Proceedings of the 1961 16th ACM national meeting*, 1961, pp. 132.101–132.104.

[27] D. Henderson, "POWER Processor-Based Systems RAS," IBM, September 2020.

[28] P. H. Hochschild, P. Turner, J. C. Mogul, R. Govindaraju, P. Ranganathan, D. E. Culler, and A. Vahdat, "Cores that don't count," in *Proceedings of the Workshop on Hot Topics in Operating Systems*, ser. HotOS '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 9–16.

[29] M.-Y. Hsiao, "A Class of Optimal Minimum Odd-weight-column SEC-DED Codes," *IBM Journal of Research and Development*, vol. 14, no. 4, pp. 395–401, 1970.

[30] M.-Y. Hsiao and J. T. Tou, "Application of Error-Correcting Codes in Computer Reliability Studies," *IEEE Transactions on Reliability*, vol. R-18, no. 3, pp. 108–118, 1969.

[31] R. Huang and G. E. Suh, "IVEC: off-chip memory integrity protection for both security and reliability," in *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ser. ISCA '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 395–406.

[32] Y. Jang, J. Lee, S. Lee, and T. Kim, "SGX-Bomb: Locking Down the Processor via Rowhammer Attack," in *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, ser. SysTEX'17. New York, NY, USA: Association for Computing Machinery, 2017.

[33] JEDEC, *JESD79-4, DDR4 SDRAM*, JEDEC Solid State Technology Association, July 2014.

[34] ——, *JESD79-5, DDR5 SDRAM*, JEDEC Solid State Technology Association, July 2020.

[35] ——, *JESD238 HBM3 Standard*, JEDEC Solid State Technology Association, January 2023.

[36] X. Jian, H. Duwe, J. Sartori, V. Sridharan, and R. Kumar, "Low-Power, Low-Storage-Overhead Chipkill Correct via Multi-Line Error Correction," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '13, Denver, Colorado, 2013.

[37] X. Jian and R. Kumar, "Adaptive Reliability Chipkill Correct (ARCC)," in *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*, 2013, pp. 270–281.

[38] J. Juffinger, L. Lamster, A. Kogler, M. Eichlseder, M. Lipp, and D. Gruss, "CSI:Rowhammer – Cryptographic Security and Integrity against Rowhammer," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023, pp. 1702–1718.

[39] 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.

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

[41] D. Kim, J. Lee, W. Jung, M. Sullivan, and J. Kim, "Unity ECC: Unified Memory Protection Against Bit and Chip Errors," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '23. New York, NY, USA: Association for Computing Machinery, 2023.

[42] J. Kim, M. Sullivan, and M. Erez, "Bamboo ECC: Strong, safe, and flexible codes for reliable computer memory," in *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, Burlingame, California, USA, Feb. 2015, pp. 101–112.

[43] M. Kounavis, D. Durham, S. Deutsch, K. Matusiewicz, and D. Wheeler, "The MAGIC Mode for Simultaneously Supporting Encryption, Message Authentication and Error Correction," Cryptology ePrint Archive, Paper 2020/1460, 2020.

[44] L. Lamster, M. Unterguggenberger, D. Schrammel, and S. Mangard, "HashTag: Hash-based integrity protection for tagged architectures," in *32nd USENIX Security Symposium (USENIX Security 23)*. Anaheim, CA: USENIX Association, Aug. 2023, pp. 2797–2814.

[45] ——, "Voodoo: Memory Tagging, Authenticated Encryption, and Error Correction through MAGIC," in *33rd USENIX Security Symposium (USENIX Security 24)*. Philadelphia, PA: USENIX Association, Aug. 2024, pp. 7159–7176.

[46] L. A. Lastras-Montaño, P. J. Meaney, E. Stephens, B. M. Trager, J. O'Connor, and L. C. Alves, "A new class of array codes for memory storage," in *2011 Information Theory and Applications Workshop*, 2011, pp. 1–10.

[47] 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.

[48] S. Levy, K. B. Ferreira, N. DeBardeleben, T. Siddiqua, V. Sridharan, and E. Baseman, "Lessons Learned from Memory Errors Observed Over the Lifetime of Cielo," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018, pp. 554–565.

[49] Y.-C. Liu, "Byte Error Correction in Memory and Arithmetic Units," Ph.D. dissertation, Northwestern University, 1970.

[50] Y. Luo, S. Govindan, B. Sharma, M. Santaniello, J. Meza, A. Kansal, J. Liu, B. Khessib, K. Vaid, and O. Mutlu, "Characterizing Application Memory Error Vulnerability to Optimize Datacenter Cost via Heterogeneous-Reliability Memory," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2014, pp. 467–478.

[51] E. Manzhosov, A. Hastings, M. Pancholi, R. Piersma, M. T. I. Ziad, and S. Sethumadhavan, "Revisiting Residue Codes for Modern Memories," in *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2022, pp. 73–90.

[52] F. McKeen, I. Alexandrovich, A. Berenzon, C. V. Rozas, H. Shafi, V. Shanbhogue, and U. R. Savagaonkar, "Innovative Instructions and Software Model for Isolated Execution," in *Proceedings of the 2nd International Workshop on Hardware and Architectural Support for Security and Privacy*, ser. HASP '13. New York, NY, USA: Association for Computing Machinery, 2013.

[53] J. Meza, Q. Wu, S. Kumar, and O. Mutlu, "Revisiting Memory Errors in Large-Scale Production Data Centers: Analysis and Modeling of New Trends from the Field," in *2015 45th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2015, pp. 415–426.

[54] P. J. Nair, D.-H. Kim, and M. K. Qureshi, "ArchShield: Architectural Framework for Assisting DRAM Scaling by Tolerating High Error Rates," in *Proceedings of the 40th Annual International Symposium on Computer Architecture*, ser. ISCA '13. New York, NY, USA: Association for Computing Machinery, 2013, pp. 72–83.

[55] P. J. Nair, V. Sridharan, and M. K. Qureshi, "XED: Exposing On-Die Error Detection Information for Strong Memory Reliability," in *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, 2016, pp. 341–353.

[56] D.-T. Nguyen, N.-M. Ho, W.-F. Wong, and I.-J. Chang, "OBET: On-the-Fly Byte-Level Error Tracking for Correcting and Detecting Faults in Unreliable DRAM Systems," *Sensors*, vol. 21, no. 24, 2021.

[57] M. Nocker, D. Drexel, M. Rader, A. Montuoro, and P. Schöttle, "HE-MAN - Homomorphically Encrypted MAchine Learning with ONnx Models," in *Proceedings of the 2023 8th International Conference on Machine Learning Technologies*, ser. ICMLT '23. New York, NY, USA: Association for Computing Machinery, 2023, pp. 35–45.

[58] Oracle, "Hardware-assisted checking using Silicon Secured Memory (SSM)," https://docs.oracle.com/cd/E37069_01/html/E37085/gphwb.html, 2015.

[59] A. Patil, V. Nagarajan, R. Balasubramonian, and N. Oswald, "Dvé: Improving DRAM Reliability and Performance On-Demand via Coherent Replication," in *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021, pp. 526–539.

[60] O. C. Project, *OCP Fault Management Infrastructure Requirements*, Open Compute Project, 2023.

[61] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the society for industrial and applied mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[62] G. Saileshwar, P. J. Nair, P. Ramrakhyani, W. Elsasser, and M. K. Qureshi, "Synergy: Rethinking secure-memory design for error-correcting memories," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, Vienna, Austria, 2018, pp. 454–465.

[63] C. Schoeny, F. Sala, M. Gottscho, I. Alam, P. Gupta, and L. Dolecek, "Context-Aware Resiliency: Unequal Message Protection for Random-Access Memories," *IEEE Transactions on Information Theory*, vol. 65, no. 10, pp. 6146–6159, 2019.

[64] M. Seaborn and T. Dullien, "Exploiting the DRAM rowhammer bug to gain kernel privileges," *Black Hat*, vol. 15, p. 71, 2015.

[65] C. E. Shannon, "Communication theory of secrecy systems," *The Bell system technical journal*, vol. 28, no. 4, pp. 656–715, 1949.

[66] I. Silicon Integration Initiative, "The Nangate Open Cell Library 45nm FreePDK." [Online]. Available: https://si2.org/open-cell-library

[67] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi, "Memory Errors in Modern Systems: The Good, The Bad, and The Ugly," in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '15. New York, NY, USA: Association for Computing Machinery, 2015, pp. 297–310.

[68] M. B. Sullivan, M. T. I. Ziad, A. Jaleel, and S. W. Keckler, "Implicit Memory Tagging: No-Overhead Memory Safety Using Alias-Free Tagged ECC," in *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ser. ISCA '23. New York, NY, USA: Association for Computing Machinery, 2023.

[69] *Memory RAS Configuration, Rev 1.0*, Supermicro Inc., 2017.

[70] M. Taassori, R. Balasubramonian, S. Chhabra, A. R. Alameldeen, M. Peddireddy, R. Agarwal, and R. Stutsman, "Compact Leakage-Free Support for Integrity and Reliability," in *Proceedings of the ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, ser. ISCA '20, Virtual Event, 2020, pp. 735–748.

[71] A. Tatar, R. K. Konoth, E. Athanasopoulos, C. Giuffrida, H. Bos, and K. Razavi, "Throwhammer: Rowhammer Attacks over the Network and Defenses," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 213–226.

[72] A. N. Udipi, N. Muralimanohar, R. Balsubramonian, A. Davis, and N. P. Jouppi, "LOT-ECC: Localized and tiered reliability mechanisms for commodity memory systems," in *2012 39th Annual International Symposium on Computer Architecture (ISCA)*, Portland, Oregon, USA, 2012, pp. 285–296.

[73] H. Venugopalan, K. Goswami, Z. A. Din, J. Lowe-Power, S. T. King, and Z. Shafiq, "Centauri: Practical Rowhammer Fingerprinting," 2023.

[74] "Armv8.5-A: Memory Tagging Extension," ARM Limited, Tech. Rep., 2018.

[75] "H3C G5 Servers RAS. Technology White Paper," New H3C Technologies Co., Tech. Rep., 2023.

[76] "Intel Trust Domain Extensions," Intel Corporation, Tech. Rep., February 2023, White Paper.

[77] Zama, "Concrete: TFHE Compiler that converts python programs into FHE equivalent," 2022, https://github.com/zama-ai/concrete.

[78] D. Zhang, V. Sridharan, and X. Jian, "Exploring and Optimizing Chipkill-Correct for Persistent Memory Based on High-Density NVRAMs," in *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2018, pp. 710–723.

[79] D. Zivanovic, P. E. Dokht, S. Moré, J. Bartolome, P. M. Carpenter, P. Radojković, and E. Ayguadé, "DRAM Errors in the Field: A Statistical Approach," in *Proceedings of the International Symposium on Memory Systems*, ser. MEMSYS '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 69–84.

[80] K. A. Zubair, S. Gurumurthi, V. Sridharan, and A. Awad, "Soteria: Towards Resilient Integrity-Protected and Encrypted Non-Volatile Memories," in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO '21. New York, NY, USA: Association for Computing Machinery, 2021, pp. 1214–1226.