HashTag: Hash-based Integrity Protection for Tagged Architectures

Lukas Lamster, Martin Unterguggenberger, David Schrammel, and Stefan Mangard, *Graz University of Technology*

https://www.usenix.org/conference/usenixsecurity23/presentation/lamster

This paper is included in the Proceedings of the 32nd USENIX Security Symposium.

August 9–11, 2023 • Anaheim, CA, USA

978-1-939133-37-3

Open access to the Proceedings of the 32nd USENIX Security Symposium is sponsored by USENIX.

HashTag: Hash-based Integrity Protection for Tagged Architectures

Lukas Lamster, Martin Unterguggenberger
David Schrammel, Stefan Mangard
Graz University of Technology

Abstract

Modern computing systems rely on error-correcting codes to ensure the integrity of DRAM data. Linear checksums allow for fast detection and correction of specific error patterns. However, they do not offer sufficient protection against complex errors distributed over multiple data words and chips. Depending on the code and the error pattern, linear codes may fail to detect or even miscorrect errors, thus leading to silent data corruption.

In this work, we show how compact error-correcting codes based on low-latency hashing functions allow for strong probabilistic error detection and correction while facilitating ECC bit repurposing. Our proposed design drastically lowers the expected rate of undetected errors, regardless of the underlying error patterns.

By tailoring the size of our codes to the required level of integrity protection, we are able to free bits that would otherwise be required to store ECC data. We showcase how our design facilitates the efficient implementation of tagged memory architectures such as CHERI, ARM MTE, and SPARC ADI by repurposing the freed bits in commodity ECC DRAM.

Thus, we harden systems against data corruption due to DRAM faults while *simultaneously allowing for memory tagging without introducing additional memory accesses*. We present a systematic analysis of schemes that allow memory tagging on a cache line granularity while maintaining error detection and correction capabilities, even in multi-bit fault scenarios. We evaluate our integrity protection with tagging for different use cases and show that we can store 32 bits of additional tags per cache line, twice the amount needed to implement ARM's MTE, without significantly affecting error correction capabilities. We also show how up to 51 bits can be made available while maintaining single-bit error correction.

1 Introduction

DRAM modules hold a variety of system-critical and sensitive data during runtime. Clearly, the integrity of this data is essential, and integrity violations threaten the overall security of computing systems. Unfortunately, data stored in DRAM memory is subject to faults. Naturally ocurring leakage or the impact of charged particles may spontaneously alter the data stored in memory. Error-correcting codes (ECC) aim to provide integrity in the presence of errors through additional redundancy, thus allowing for error detection and correction. In ECC DRAM modules, redundant information is stored in a dedicated chip and fetched in parallel with the associated data. Each memory read validates the data integrity and corrects errors if needed. In commodity solutions, linear codes deterministically protect data against a fixed upper limit of erroneous bits. Errors that exceed the capabilities of the deployed code, however, may remain undetected or even cause miscorrection, thus violating data integrity.

In addition to naturally ocurring DRAM errors, memory safety issues are a prevalent threat to all computing systems. Microsoft [50] and Google [34] independently claim that memory safety errors account for approximately 70 % of all bugs in their products. Adversaries exploiting memory errors can mount a variety of attacks like privilege escalation, information disclosure, or denial of service. Security has become a key ingredient of software systems and is deeply anchored into the system's hardware architecture. Novel hardware-software co-designs introduce strong hardware-backed security properties, ensuring reliable protection for the deployed software.

Memory tagging is a versatile tool utilizing single- or multi-bit tags to enforce fine-grained security policies. Various research proposals [10, 11, 28, 42, 47, 51] utilize tagged memory architectures for dynamic information flow tracking (DIFT). Capability-based architectures (e.g., CHERI [55], M-Machine [6]) utilize single-bit memory tagging to distinguish between data and capabilities in memory. Commercial products, like the ARM memory tagging extension (MTE) [26] and SPARC application data integrity (ADI) [1], utilize memory tagging to provide probabilistic memory safety on an object granularity. Besides memory safety, mechanisms like Mondrian [54] and Loki [58] utilize memory tagging for memory isolation.

Tagged memory architectures enforce security policies to target a diverse set of threat models.However, all tagged memory architectures share one common drawback: *Memory tagging requires additional DRAM requests to fetch the tags from the main memory*. This imposes additional pressure on the memory bus, thus decreasing system performance. The overhead is simultaneously the bottleneck of tagged architectures and an obstacle that needs to be overcome to facilitate widespread adoption. For example, the HDFI [42] prototype introduces DRAM access overheads ranging from 37.9 % up to 373 %. Previous work [8, 20, 55] on memory tagging highlights that ECC memory could be deployed to reduce the overhead of tagged architectures. Precisely, tags could be stored in ECC bits to eliminate the impact on the system performance due to the additional tag fetches. Utilizing ECC DRAM allows reading the data and tags in parallel, effectively avoiding multiple memory accesses. *However, it is currently unknown how repurposing ECC bits for memory tagging influences error detection and correction capabilities.*

In this paper, we introduce a compact, hash-based integrity protection scheme that allows the implementation of memory tagging without additional tag fetches on commodity ECC DRAM modules. We combine hash functions and parity bits for error detection and correction, repurposing the remaining bits to co-locate the memory tags with the corresponding data. Contrary to existing combinations of tagging and linear codes [14, 15, 31], our approach performs significantly better for multi-bit errors.

Our results show that we can implement tagging schemes using up to 16 bits per cache line while increasing error detection and correction capabilities. This tag space is large enough to implement various tagged memory architectures (e.g., CHERI, ARM MTE, and SPARC ADI) without generating any additional memory traffic. In addition, we introduce evaluation models for architectures focussing on large tag sizes. Precisely, we show that tagged architectures utilizing 32, 46, and 51 bits are implementable and discuss their tradeoffs regarding error detection and correction. Our findings underline that our method of eliminating bandwidth overheads due to tag fetches is feasible, even for architectures that use large tags. We implement our design in the gem5 simulator and evaluate the performance compared to an MTE-like tagged memory architecture. The evaluation shows that we can dramatically reduce the performance overheads that are typical for tagged memory applications.

Contributions. We make the following contributions:

- We are first to show how compact hash-based integrity protection codes can be combined with memory tagging utilizing commodity ECC DRAM modules.
- We investigate hash-based error detection and correction under real-world DRAM failure models. Our results allow us to provide up to 16-bit memory tags on a cache line granularity while supporting more robust integrity protection than commodity devices.
- We systematically analyze tagged architectures like CHERI, ARM MTE, and SPARC ADI, showing that they can be implemented without introducing additional pressure on the memory bus. We evaluate our approach and show an average reduction of overheads by a factor of approximately 20.
- We highlight the potential of novel tagged architectures facilitating larger tag sizes (up to 51 bits per cache line) and discuss the impacts on error correction and detection capabilities.

Outline. The paper is outlined as follows. Section 2 provides the necessary background. Section 3 defines fault models based on DRAM fault behavior. Section 4 describes hash-based designs, shows how hash sizes influence error detection and correction capabilities, and investigates different hash-based designs. Section 5 analyzes possible instantiations of our hash-based design. Section 6 elaborates on the overheads of our scheme with regard to performance, correction times, and area. Section 7 discusses related work, and Section 8 concludes this work.

2 Background

This section introduces the concept of memory tagging and tagged memory architectures. We discuss how tagged memory can be used to enforce memory safety or to isolate protection domains. Furthermore, we give an overview of the organization of modern DRAM devices. We discuss how faults in DRAM cells can influence the stored data and how protection mechanisms aim to detect and correct errors in DRAM.

2.1 Tagged Memory Architectures

Various modern secure hardware architectures use tagged memory as a fundamental building block [40]. Tagged architectures augment the memory stored in DRAM with additional metadata to enforce policies. While memory tagging allows for arbitrary policies, it is especially relevant for system security. Depending on the tagging scheme, different security concerns, e.g., memory safety, are addressed.

Researchers proposed various hardware architectures implementing fixed, partly configurable, and freely configurable policies through single- or multi-bit tags. A common type of memory tagging is dynamic information flow tracking (DIFT) [47], which analyzes the propagation of user input throughout the execution of the program. Architectures apply DIFT [10, 11, 28, 51] to track potential malicious user input and mitigate exploitation techniques like ROP [5, 41] and JOP [7].

Commercial solutions like ARM MTE [26] and SPARC ADI [1] focus on memory safety policies at object granularity. These architectures target different classes of memory access violations, e.g., temporal and spatial violations. On object

allocation, the corresponding memory locations are associated with a dedicated memory tag. Additionally, this tag is encoded in the unused upper bits of the referencing pointer. At each memory access, the tagged memory architecture checks whether the tag of the pointer and the corresponding tag in memory match. This procedure allows the detection of temporal and spatial memory violations on a probabilistic basis.

Capability-based architectures use tagging to enforce the integrity of capabilities in memory. Precisely, CHERI [55] and the M-Machine [6] utilize a single-bit tag per granule. This fine-granular memory protection only permits memory accesses executed from capability instructions, providing the desired integrity.

Tagged memory architectures require additional DRAM requests to fetch the tags from memory. This introduces a significant performance overhead due to the additional pressure on the memory bus [20]. To counteract this performance penalty, efficient tagged memory architectures utilize a tag store [20] serving as a separate cache for memory tags.

2.2 DRAM Organization

Commodity DRAM devices store information in *memory cells* consisting of a transistor and a capacitor. Multiple cells are grouped in *rows* and *columns*, thus forming a *memory array*. On a single DRAM chip, multiple memory arrays operate in parallel, and the number of arrays determines the bus width of the chip. Independent *DRAM channels* allow the memory controller to communicate with sets of DRAM chips in parallel, thus increasing the performance. In commodity devices, the bus width of a single DRAM chip is usually 4 or 8 bits. Depending on the bus width of the individual chips, we speak of *x4* or *x8* DRAM. Transmitting large amounts of data over such a small bus would be infeasible and slow. Hence, multiple DRAM chips are grouped, forming a *rank*. One channel can access multiple ranks, depending on the actual hardware configuration. In modern system architectures, accesses to DRAM are *burst-oriented* [46]. A single read from a DRAM channel usually yields 512 bits of data which is the typical size of a cache line. Given the bus size of a DDR4 device, we see eight *beats* of 64 bits each. Reading in bursts is more efficient than performing single word-granular accesses, as the 512 bits burst size matches the size of a cache line in most modern systems.

2.3 DRAM Errors

When discussing errors in DRAM cells, we distinguish between *faults* and *errors*. A fault is the underlying cause of memory corruption. An error occurs when reading from a faulty location and can be considered the manifestation of a fault. The charges stored in DRAM cells leak over time, thus reducing the voltage level of the capacitor. Cell contents must be refreshed periodically to avoid data loss. The *retention time* denotes the duration that a cell can hold a stored charge such that the value read from the cell equals the value that was previously stored. The JEDEC standard [46] specifies that the retention time of DRAM cells must be at least 64 ms. Even if all cells fulfill the specification, it can happen that too much charge leaks from a capacitor. Excessive leakage prevents the sense amplifiers from correctly measuring the stored charge and, thus, the logical value read from the cell changes. Research shows that different external influences, such as radiation [3] or temperature [27], can alter the stored contents of memory cells by accelerating the leakage speed.

An error in which a stored bit erroneously changes its value is called a *bit flip*. While bit flips occur due to charge leakage, logical bits may also flip from '0' to '1' depending on the interpretation of the cell's content.

A cell that experiences a *transient fault* is not necessarily broken. Instead, the fault is fixed as the cell content is overwritten. Reading from a memory location that had a transient fault in the past will, in general, not result in an error unless the location experiences another fault. Thus, transient faults are usually observed once but do not necessarily reoccur at the same location.

Hard faults are not fixable without replacing the complete DRAM module, as they occur due to physical damage in the component. Reading from a location that has experienced a hard fault will always cause an error unless the affected component is replaced.

Studies indicate that the absolute values of transient and hard errors strongly depend on the DRAM manufacturer [25, 39, 44].

2.4 DRAM Integrity Protection

In order to protect against data corruption, *error-correcting codes* (ECC) are deployed. Usually, these codes are linear codes that are implemented in hardware in the memory controller. When using ECC, redundant information is stored in additional memory next to the actual data. The storage holding the redundant information is only accessible by the memory controller. On each memory access, the memory controller uses redundant information to check for data corruption. In commodity systems, *single-error correction double-error detection* (SEC-DED) codes with 64-bit granularity are used. For DDR4 DRAM, each 64-bit data word is extended by 8 checksum bits, thus resulting in a code word size of 72 bits. Expanding the bus size and reading the parity bits in parallel with the data allows for fast checking and avoids additional fetches from memory. When a data word is read, the ECC logic computes a so-called *syndrome* over the current data and the stored parity information. In SEC-DED codes, the syndrome indicates the location of the erroneous bit in the case that an error occurred. Thus, it is possible to directly correct the faulty bits. When detecting double-bit errors, SEC-DED codes are not able to correct the error, and the memory

controller raises a *machine check exception*. If more than two errors per 64-bit word occur, SEC-DED codes cannot guarantee that the error is detected. Depending on the location of the flipped bits, it is possible that the error is not detected or wrongfully interpreted as a single-bit error. Thus, SEC-DED codes may miscorrect errors that exceed the capabilities of the code. We call such cases *silent data corruption* (SDC).

Chipkill [12] distributes the bits from one DRAM chip over multiple ECC words. Even if a full chip fails, only one bit per ECC word is affected. Thus, a complete chip failure can be detected and corrected. Contrary to SEC-DED codes, Chipkill is a symbol-based correction mechanism. As the symbol size influences the capabilities of the code, the overhead of Chipkill directly correlates to the bus width of the used DRAM chips. For x4 devices, the overhead is equal to that of SEC-DED codes, and for x8 devices, the overhead would be doubled.

Systems that use ECC may also use a technique called *memory scrubbing*. When this is enabled, the system will periodically perform read operations over the DRAM. The main goal of scrubbing is to reduce the number of uncorrectable errors by avoiding the accumulation of localized errors.

3 Defining the Fault Model

In order to compare the error detection and correction capabilities of existing and novel schemes, it is imperative to define a realistic fault model. In this section, we define the fault patterns that we consider during our analysis. We base our model on previous research on DRAM faults in large-scale systems [25, 39, 44]. While available data is limited, we identify five failure modes that range from simple single-bit errors to complex multi-bit error patterns. While there is a multitude of ways that failures can manifest in DRAM, we limit our analysis to failure modes that can actually be corrected by rank-level ECC, such as SEC-DED or Chipkill codes.

We define our failure modes depending on the way that the errors manifest at cache line granularity. Our analysis targets systems with a cache line width of 512 and a burst size of 64 bits, thus resulting in 8 beats per burst. For multi-bit failure modes, we differ between x4 and x8 configurations due to different error patterns in the case of partial or complete chip failures. Table 1 gives a concise overview of the failure modes in our fault model.

3.1 Single-bit Failure Modes

Single-bit errors are the prevalent error type in modern DRAM devices [25, 39, 44]. A single-bit error can have multiple causes. When a DRAM cell experiences a hard fault, it becomes stuck at a fixed value. Such a cell will always return the same value when accessed. Contrarily, a cell can experience a transient error due to excessive leakage, particle strikes, or other influences. In such cases, the cell will regain functionality after the subsequent write access. If an entire column in a DRAM device fails, we still see a single-bit error on access, as only one bit per column is accessed during a burst. Only reads to different rows will trigger multiple errors in the case of a column failure. Each single-bit fault, no matter the underlying cause, will manifest itself as a single erroneous bit in a cache line. SEC-DED and Chipkill codes both guarantee to detect and correct single-bit errors. We denote the single-bit failure mode as *F1*.

Table 1: The failure modes that we include in our fault model.

Failure Mode	Bits Affected	Description
F1	1	Single faulty bit
F2	8	Single stuck pin
F3S	up to 56	Multiple stuck pins in a single chip
F3M	up to 56	Multiple stuck pins in multiple chips
F4	up to 64	Broken chip (all pins stuck)
F5S	up to 57	F3S + transient fault
F5M	up to 57	F3M + transient fault

3.2 Multi-bit Failure Modes

While faults that affect multiple bits are rare, we must consider them when analyzing error detection and correction mechanisms. The least complex multi-bit error pattern is observed when a single pin of one DRAM chip becomes stuck at a fixed value. Contrary to the single-bit fault mode in which a single cell is stuck, a compromised pin will affect every word read from the stuck DRAM chip. The number of corrupted bits is equal to the number of beats per burst. In each beat, a single word with a stuck bit is read from the faulty chip. Thus, a stuck pin will cause 8 bits in the cache line to equal the stuck value for both x4 and x8 chips. With a 64-bit burst size, all corrupted bits are expected to be exactly 64 bits apart. This failure mode is detectable and correctable by SEC-DED and Chipkill codes, as the single erroneous bit per 64-bit word stems from a single chip. We refer to this failure mode as *F2*.

A more complex failure mode occurs when multiple pins are stuck. We distinguish between the case that multiple pins from the same chip are stuck (*F3S*) and the case in which pins from multiple chips are stuck (*F3M*). In both cases, the number of corrupted bits per cache line equals the number of stuck bits multiplied by the number of beats per burst. This failure mode is similar to the single stuck pin mode in the aspect that all bits in the cache line that originate from a stuck pin have the same value. As the number of corrupted bits per 64-bit word exceeds the correction capability of SEC-DED codes, correction becomes impossible. Depending on the number of affected pins, a SEC-DED code might be unable to correctly identify a multi-pin failure, thus resulting in silent data corruption or miscorrection. While the F3S failure mode is correctable by Chipkill codes, F3M is, in general, not correctable. When discussing F3S and F3M failure modes,

we must consider the chip size and the number of stuck pins. We use the notation F3S(s,f) and F3M(f) to denote F3 failure modes with an s-bit chip size and up to f stuck pins.

In the case that a complete chip failure occurs, we must also distinguish between x4 and x8 configurations. When using x4 DRAM chips, the complete failure of one chip results in 32 corrupted bits in the worst case. Systems that use Chipkill codes can correct such chip failures in the case of x4 chips [16]. As x8 chips contribute more bits per cache line, complete chip failures become more severe. When an x8 chip fails, up to 64 bits per cache line are corrupted. Thus, error correction becomes more complicated and is not supported by all Chipkill codes. We denote the case of a complete chip failure as *F4*. Note that F4 is an edge-case of the F3S failure mode in which all pins are considered stuck. As we need to differ between x4 and x8 configurations, we denote F4 failures as F4(s) to include the chip size in our notation.

3.3 Combined Failure Modes

In theory, it is possible that multiple failure modes occur at the same time, thus causing the corruption of many bits at once. Complex combinations of failure modes are, however, rare [4].

We only consider cases in which a single transient fault occurs in parallel with a hard fault. A single additional transient fault will cause most Chipkill schemes to fail if it occurs in a device that is different from the one that experiences the stuck bit. In cases where the transient fault is co-located to the hard fault, i.e., occurring in the same chip, Chipkill codes can detect and correct the fault. For SEC-DED codes, this fault pattern poses a problem. A combination of a single-pin hard fault and a single transient fault will be detected but not corrected. Should a transient fault occur next to a multi-pin fault, the SEC-DED error detection will fail, and an undetected error will occur.

As an F3M fault will not be corrected by Chipkill, the combination of a transient fault with an F3S fault will also be uncorrectable. We denote the combination of transient failures with F3 failures as *F5*. The F5 failure mode inherits the notation from the F3 mode. For the combination of F3S and a transient fault, we assume that the transient fault occurs in a different chip from the one with the stuck pins.

We do not include transient multi-bit faults in our analysis as they are unlikely to occur [4]. We do not consider glitches and faults of the command and address signals in our fault model as they are assumed to be protected by an orthogonal measure. Furthermore, we do not consider faults induced by Rowhammer attacks in our model. Such faults imply the presence of an active attacker and are best thwarted by dedicated Rowhammer mitigations [18, 32, 56, 57].

3.4 Fault Rate

When discussing fault rates, it is practical to use *failures in time* (FIT). One FIT equals one failure event per 10^9 hours of operation. To accurately model real-world fault behavior, we use the fault rates reported by large-scale studies on DRAM fault behavior [43–45]. Unfortunately, DRAM fault rates vary strongly across different memory modules and vendors. The reported *overall fault rates* range from 73.6 FIT/DRAM to 18.8 FIT/DRAM in [43]. Furthermore, Sridharan et al. [44] find the largest rate of SEC-DED *undetectable faults* to be 21.7 FIT/DRAM, while the lowest rate is given with 0.2 FIT/-DRAM.

We estimate the expected fault rate based on the empirical fault rates reported in [43–45]. In their work, they list the overall rate of faults per DRAM as well as the rate of undetected faults per DRAM. We compute the overall expected rate of faults by averaging the reported overall fault rates per DRAM device. Likewise, we compute the expected rate of undetectable faults by averaging the given numbers of undetectable faults per DRAM. Thus, we reach an expected fault rate of 45.32 FIT/DRAM and an expected rate of undetectable faults of 7.9 FIT/DRAM. We denote the expected undetectable fault rate as R_{UD} and the expected overall fault rate as R.

4 Error Detection and Correction for Hash-based Integrity Protection

This section provides an in-depth analysis of hash-based integrity protection schemes and their error detection and correction capabilities. First, we identify the fundamental metrics that shape each of our hash-based schemes. Then, we discuss the principles of hash-based error detection and correction. We elaborate on the problems and pitfalls of error correction that we face when moving away from linear codes. Then, we show how additional parity bits and information about common fault patterns can be leveraged to greatly increase the correction performance of hash-based schemes. Concluding this section, we discuss the case in which the hash itself becomes corrupted.

We identify three metrics that require special consideration when designing a hash-based integrity protection scheme.

Detection and Correction Capabilities. While it would be trivial to repurpose ECC bits as tag bits, doing so without losing the ability to detect and correct errors is not a straightforward task. Balancing the number of tag bits and the number of hash bits such that an optimum between tag size and integrity protection capabilities is obtained is crucial. The failure modes and fault rates defined in Section 3 constitute the basis of our analysis. The expected rate of undetected faults plays a vital role as it allows us to compare hash-based schemes against linear codes. Guaranteeing the detection of

common failure modes is important for systems that require high availability.

Latency and Overheads. When using a hash-based approach, the latency that the hash circuitry introduces can lead to performance degradation. We thus look for designs that allow for efficient hardware implementations and, subsequently, lower latencies. The goal is to keep the performance loss due to additional computation time low. This is especially important when considering hashes based on block ciphers that take multiple cycles for each computation to complete. While the hash latency influences each memory request, the correction latency influences system performance only when failures occur. Thus, frequently occurring fault patterns must be handled efficiently. Long correction latencies for rare failure modes, however, are not as problematic.

Tag Size and Granularity. The possible tag sizes and granularities are the limiting factors for the tagged memory architectures that a hash-based scheme can implement. We search for fine-granular tagging with the largest achievable tag size while still providing sufficient detection and correction capabilities. This metric directly opposes the detection and correction capabilities. Large tag sizes allow for powerful tagging policies but reduce the number of available redundancy bits, thus influencing integrity protection negatively.

4.1 Hash-based Integrity Protection

In general, hashes allow us to calculate a fixed-size fingerprint of a variably sized input. Changing a single bit in the input will lead to a large difference between the resulting outputs. Thus, hash functions are a feasible way of detecting data manipulation.

Settings in which an active attacker tries to force a collision of the checksum values by introducing targeted errors in the input data require the use of a *message authentication code* (MAC) or a *cryptographic* hash function. Juffinger et al. [21] and Fakhrzadehgan et al. [13], for example, show how MAC functions protect against an adversary performing Rowhammer attacks. According to the fault model defined in Section 3, we only consider naturally occurring DRAM faults. Under this model, the distribution of errors is not targeted but random. Thus, it is sufficient to select a function that generates checksums such that the probability of two inputs having the same checksum is low. It is not required to use a function that offers cryptographic security. Furthermore, our use case does not require a function that can calculate a hash over an arbitrarily sized input. Instead, the input size is known and equal to the size of the data over which we compute the checksum.

Detecting errors. Assume a function $\mathcal{H}$ that maps n input bits to k output bits such that the output is drawn uniformly from the $\{0,1\}^k$ output space. The probability that an input D' maps to the same checksum as a different input D is then equal to 2^{-k}. An error detection scheme using such a function misses an error only if the erroneous input maps to the same output as the original input. Thus, each occurring error, independent of the error pattern and the number of faulty bits, has a probability of $P(\text{undetected}) = 2^{-k}$ to be undetected.

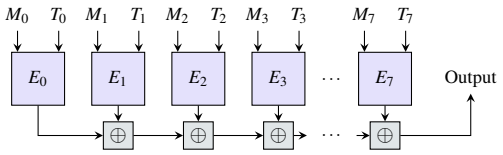

Figure 1: A hash that is computed from parallel block cipher invocations that are XOR-ed together. To fit additional data, the final output is truncated.

Correcting errors. While such a function allows for error detection, it poses a problem for error correction. Unlike with linear codes, it is not possible to directly compute the location of erroneous bits from the difference between two checksums. Thus, probabilistic integrity protection schemes usually rely on brute-force search for error correction [13, 17, 21]. Given the function H, a faulty input D', and the correct input D, the error correction mechanism aims to find a binary vector E such that $\mathcal{H}(D' \oplus E) = \mathcal{H}(D)$. Without additional knowledge of the location and distribution of bit errors in D, the best possible approach is to try all values of E in a brute-force search.

Constructing a suitable function. Our design allows the use of any function that fulfills the properties described above. In our analysis, we investigate two functions based on the tweakable block cipher `QARMA` [2] and the low-latency block cipher `SPEEDY` [24]. A block cipher maps an n-bit input, a K-bit key to an n-bit output. A *tweakable* block cipher processes a T-bit tweak in addition to the input. The output of both variants is uniformly distributed. When summing the outputs of multiple independent block cipher instances using the XOR operation, the resulting value is, again, uniformly distributed.

Figure 1 shows a completely parallelizable structure that fulfills the previously described properties. As each 512-bit fetch from DRAM is divided into 8 64-bit messages, we use the same structure for our construct. Each cipher instance (e.g., `QARMA`) processes a 64-bit input (M_0 to M_7) and generates a corresponding 64-bit output. The tweaks are chosen such that they are unique for each instance, i.e. $T_i \neq T_j$ holds. By doing so, we ensure that the cipher texts of two equivalent message blocks do not cancel each other out in the XOR operation. Note that this construct *does not yield cryptographic security against an attacker* [52]. The presented structure is, thus, not a cryptographic hash function. However, our setting only considers naturally ocurring DRAM errors. Thus, it is sufficient to focus solely on the probability of two checksums colliding.

Figure 2 illustrates a PMAC design built from tweakable block ciphers [36]. Contrary to the design in Figure 1, the PMAC offers cryptographic security against an attacker.

Hence, one could use this construct to protect DRAM integrity, even in the presence of an adversary [21]. The additional security, however, comes with a cost. Due to the structure of the PMAC the latency is increased as the resulting output can only be computed once all other intermediate computations have finished. Assuming that the inputs M_0 to M_7 arrive in sequence, this effectively doubles the latency, as the last block cipher invocation cannot be computed in parallel.

Figure 3 shows a similar PMAC design but built from the SPEEDY block cipher. SPEEDY uses 192-bit inputs and outputs but does not support additional tweaks as inputs. Hence, to ensure different ciphertexts for same messages, we have to use different keys K_i for each cipher instance. The final cipher instance E_2 takes two messages as input, which leaves 64 bits for additional data (AD). A completely parallelizable (non-PMAC) structure can also be built with three SPEEDY instances analogous to Figure 1.

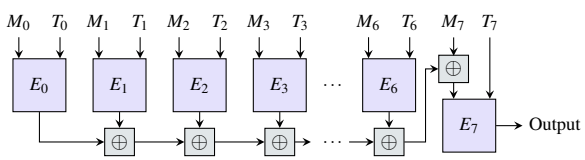

Figure 2: A PMAC construction as used by Juffinger et al. [21]. It provides cryptographical security but imposes additional latency.

We investigate the performance implications for both models in Section 6. Note that the presented structures are not exhaustive. Our proposed design allows for a wide variety of functions, such as cryptographic hashes, general-purpose hash functions, or any other function that offers a uniformly distributed output.

4.2 Reliability and Error Correction

When comparing hash-based integrity protection to commodity ECC implementations, we need a metric that allows us to argue about the quality of the protection scheme. Undetected errors threaten system integrity as corrupted data is unknowingly processed. We decide on the rate of undetected errors as the metric with which we compare our scheme to existing implementations.

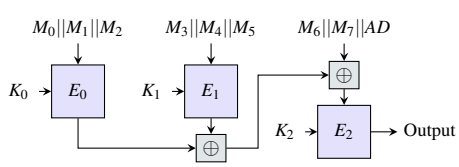

Figure 3: A PMAC construction using the 192-bit SPEEDY block cipher [24].

For a hash-based code to perform better than a commodity implementation, the hash-based variant has to experience a lower rate of undetected errors. Note, however, that comparing the raw collision probability (2^{-k} for a k-bit checksum) with the rate of undetectable errors would be an oversimplification and bias the result. Instead, we have to account for the fact that a collision only leads to an undetected error if the input of the hash function is corrupted. Arbitrary collisions between unrelated inputs can be ignored as they do not compromise the detection capabilities in the error case.

Finding the expected rate of undetected errors. Let R denote the overall DRAM fault rate and R_{UD} the rate of undetected faults as defined in Section 3. With a hash-based approach, each access to data causes the computation of a checksum. The probability that the checksum of a faulted data block coincides with the checksum of the original error-free data is equal to 2^{-k}, given a k-bit hash output. Thus, the expected rate of undetectable faults for a hash-based approach, denoted by $R_{UD,H}$, is equal to $R \cdot 2^{-k}$. If a hash function with k output bits fulfills the condition,

$$R \cdot 2^{-k} \leq R_{UD} \tag{1}$$

then the hash-based approach has an expected undetected error rate that is at most R_{UD}.

We can use this observation combined with the fault rates given in Section 3 to compute a numerical lower bound of the output size k. By solving for k, we reach the condition

$$k \geq \mathrm{ld}\left(\frac{R}{R_{UD}}\right). \tag{2}$$

The number of bits needed to provide a similar undetected error rate as the state-of-the-art can be calculated as

$$45.32 \cdot 2^{-k} \leq 7.9 \tag{3}$$

and yields a required output size of $k > 2$ bits.

Accounting for error correction. Given the relatively high number of faults that occur in commodity DRAM devices, the ability to correct errors is essential for system availability. Thus, a hash-based integrity protection scheme must support error correction as well. When using a correction methodology like the one presented by Saileshwar et al. [37], the number of trial computations acts as a scaling factor for the reliability. With an increasing number of trial computations, the probability of a checksum collision and, thus, of a miscorrection, increases as well.

Assume the occurrence of an f-bit error. Finding f erroneous bits in a cache line is equal to selecting f bits out of all 512 available bits. For each selection of f bits, one applies the selected error pattern to the data by XORing the mask onto the data. Thus, the worst-case number of trial computations for *exactly* f errors is $\binom{512}{f}$. We will denote the worst-case number of computations as $\mathcal{C}$.

Table 2: The number of correctable bits f strongly influences the needed hash size k. $\mathcal{C}$ denotes the worst-case number of computations for a cumulative error correction as shown in Equation (4).

f	1	2	3	4	5	6	7	8	9
$\mathcal{C}$	2^9	2^{18}	2^{25}	2^{32}	2^{39}	2^{45}	2^{51}	2^{57}	2^{63}
k	12	20	27	34	41	48	54	60	65

When considering error correction for *up to* f errors, a cumulative number of trial computations can be given as $\sum_{i=1}^{f}\binom{512}{i}$. In our analysis, we assume that each fault causes the worst-case number of trial computations during correction. We modify Inequality 3 to reflect the additional computations, yielding Inequality 4.

$$45.32 \cdot \sum_{i=1}^{f}\binom{512}{i} \cdot 2^{-k} \leq 7.9 \tag{4}$$

Solving Inequality 4 for k allows us to compute the minimal checksum size needed to correct up to f bits on a cache line granularity.

Table 2 shows how the number of correctable bits and the needed hash size correlate. Note that we give $\mathcal{C}$ as a power-of-two, as this format is more intuitive when estimating the correction latency. The given values represent the iteration count for correcting *up to* f bits in a 512-bit cache line.

In commodity DDR4 ECC memory modules, the number of available ECC bits per cache line is fixed to 64. Thus, we have to balance the number of correctable bits influences against the maximum available tag bits. Limiting the correction to fewer bits per cache line allows the usage of a smaller hash and, thus, increases the number of tag bits.

The number of trial computations is an important metric for the error correction procedure. It allows us to estimate the feasibility of each configuration as it directly influences the correction latency. While it is possible to correct up to eight randomly distributed errors per cache line using a 60-bit hash, performing almost 2^{57} computations is not realistic. The estimated correction time can be computed as the product of the hash latency and $\mathcal{C}$. Even if each hash computation and check only takes 0.3 ns, which equals the minimum latency of the `SPEEDY` block cipher, correcting an 8-bit error would take approximately 500 days. Table 2 lists the cumulative iteration count C for the correction of up to f faults. Gray cells indicate that the correction latency would exceed 1 s, assuming that a single hash computation takes 0.3 ns. Thus, up to three errors are correctable within a reasonable amount of time using a naïve brute-force approach.

4.3 Improving Error Correction

Clearly, it is necessary to improve the number of realistically correctable bits to reach a correction performance comparable to commodity SEC-DED or Chipkill solutions. A straightforward approach for reducing the correction latency is to reduce the latency of the hash computation. Unfortunately, this approach does not scale well, as the hash latency is just a linear factor in the correction latency. Each additional correctable bit will increase $\mathcal{C}$, and, thus, the correction latency, approximately exponentially.

The superior approach is to reduce the number of trial computations by reducing the growth rate of $\mathcal{C}$. We investigate two solutions with which $\mathcal{C}$ can be effectively reduced. First, we show how additional parity bits influence the error correction behavior. Then we discuss how common fault patterns play a role when performing error correction in hash-based systems.

Improving error correction through parity bits. As the number of required hash bits is fairly low, one can use the free bits to implement additional parity checks over the data. Parity bits improve the correction capabilities as they allow to estimate the location of erroneous bits in certain cases. We call schemes that add parity bits to hash-based integrity protection *parity-assisted* schemes.

One has to decide on the number of parity bits and on the granularity over which the parity is computed. The most primitive form of parity-assisted integrity protection introduces one additional parity bit that is calculated over the 512 data bits. With one additional bit, the error correction can distinguish between even and odd numbers of bit flips. Through this simple modification, single-bit error detection is guaranteed.

Increasing the number of parity bits allows us to further improve the error detection and correction capabilities. Assume, for example, a system in which the hash is assisted by 8 parity bits. Each parity bit is computed over a 64-bit word, thus dividing the cache line into eight blocks. With this modification, all single-bit errors and all multi-bit errors that are limited to one error per 64-bit word are detected. As each 64-bit block has its own parity bit, the brute-force correction only needs to consider blocks with a parity mismatch. Thus, the correction complexity is reduced by narrowing down the search space to the blocks that actually show an error.

As the number of parity bits increases, they each cover a smaller block size. It is reasonable to assume that each parity bit covers the same number of bits. Thus, the number of blocks is equal to the number of parity bits.

Fault pattern dependency. For multi-bit fault modes, the feasibility of the additional parity bits greatly depends on the fault pattern. Parity bits can only detect an odd number of errors per block, as an even number will cancel out due to linearity. Thus, the effect of additional parity bits is maximized if only a single error per block occurs. In our analysis, we pay special consideration to the worst-case number of trial computations as it is decisive when determining the number of correctable bits. We thus search for fault patterns that cause no parity mismatch and maximize the needed number of trial computations.

Two simple observations form the basis of our analysis. *(i) A block can only be corrupted without causing a parity mismatch if the number of corrupted bits in the block is even. (ii) In the case of an odd-numbered error, at least one block will experience a parity mismatch.*

For even-numbered errors, the parity bits cannot indicate the error location. The only additional information that they contribute is the fact that the number of errors must be a multiple of two. When searching for even-numbered error patterns, the correction algorithm can skip all odd-numbered patterns. The number of trial computations will ultimately depend on the order in which the correction algorithm tries different error masks. A reasonable approach would be to start by applying all 2-bit error masks to each block subsequently. Should the error still persist, the algorithm can increase the mask size to the next even number of errors. For error masks that include more than two bits, the correction procedure must also test distributions in which two or more blocks are erroneous. Thus, the correction will only perform slightly better than the naïve brute-force approach.

An odd-numbered error will cause a parity mismatch in at least one block. Thus, the correction procedure can start the correction by testing all single-bit errors in the faulty block. If no correct error pattern is found, the mask size is increased to the next odd number. As with even errors, odd-numbered error correction must also consider error distributions that affect two or more blocks.

Finding the worst-case number of trial computations. Let $p \in \{4, 8, 16\}$ denote the number of parity bits. Assume a configuration in which p parity bits are computed over p blocks where each block contains $n = 512/p$ bits. Let f denote the number of faulty bits in the cache line.

First, we consider the case that f is even and that the faulty bits are distributed such that no parity mismatch occurs. The correction algorithm will initially apply all 2-bit error masks to each block subsequently, resulting in $\binom{n}{2} \cdot p$ computations. If no matching error pattern is found, the algorithm will increase the mask size to 4. The algorithm will have to check all combinations that distribute 4 errors over p blocks such that each block experiences an even number of errors. Testing all 4-bit error masks results in $\binom{n}{4} \cdot p$ additional computations. Furthermore, all pairwise combinations of 2-bit errors are checked as well, resulting in $\binom{n}{2}^2 \cdot \binom{p}{2}$ computations. The overall worst-case computation count for 4-bit errors equals the sum of both variants.

The reasoning for odd-numbered errors follows a similar pattern. For a single erroneous bit, $\mathcal{C}$ equals n. Increasing the number of errors allows for more possible distributions where one block experiences a parity mismatch. For three errors, the correction algorithm will initially try all 3-bit error masks in the block with the parity mismatch. Then, all distributions where the mismatching block experiences one error and the two remaining errors affect a different block have to be tested. Thus, $\mathcal{C}$ can be given as $\binom{n}{3} + \binom{n}{2} \cdot (p-1) \cdot n$.

Table 3: The formula for the number of parity-assisted trial computations for even-numbered errors (f).

f	$B(f,p,n)$
2	$\binom{n}{2}p$
4	$\binom{n}{4}p + \binom{n}{2}^2\binom{p}{2}$
6	$\binom{n}{6}p + 2\binom{n}{4}\binom{n}{2}\binom{p}{2} + \binom{n}{2}^3\binom{p}{3}$
8	$\binom{n}{8}p + 2\binom{n}{6}\binom{n}{2}\binom{p}{2} + \binom{n}{4}^2\binom{p}{2} + 3\binom{n}{4}\binom{n}{2}^2\binom{p}{3} + \binom{n}{2}^4\binom{p}{4}$

As the number of erroneous bits increases, so does the number of ways to distribute them over the available blocks. Even with additional parity bits, the growth rate of $\mathcal{C}$ is high.

Let $B(f,p,n)$ be the formula that computes $\mathcal{C}$ for a certain set of parameters. For even-numbered errors, we give the explicit formula as listed in Table 3.

For odd-numbered errors, the formula can be given recursively as

$$B(f,p,n) = \binom{n}{f} + \sum_{i=1}^{\lfloor f/2 \rfloor} \binom{n}{f-2i} \cdot B(2i, p-1, n). \quad (5)$$

Table 4 shows the resulting *cumulative* values of $\mathcal{C}$ for different configurations of parity bits and faults. We reach these values by summing up the number of iterations for correcting *up to* f errors. While parity bits significantly help to reduce the computation count, the worst-case number of computations is still high. For odd-numbered errors, the performance gain is slightly more significant than for the even-numbered case. The values that are depicted in gray indicate that the correction takes longer than 1 s, even when using a state-of-the-art low-latency block cipher like SPEEDY [24].

4.4 Common Fault Patterns

While parity bits help to increase the feasibility of certain fault patterns, the error correction procedure can still take a long time. Also, the requirements for the size of the hash limit the possible combinations with memory tagging approaches. Decreasing $\mathcal{C}$ further by adding additional parity bits is infeasible as the number of bits for hash and memory tags will become too small. However, we suggest a different approach

Table 4: The worst-case *cumulative* trial computations $\mathcal{C}$, shown as power of twos, needed for error correction in the parity-assisted scheme. f denotes the number of faults, while the rows show the required number of parity bits p.

p \ f	1	2	3	4	5	6	7	8
4	2^7	2^{15}	2^{19}	2^{29}	2^{33}	2^{42}	2^{47}	2^{54}
8	2^6	2^{14}	2^{16}	2^{27}	2^{30}	2^{39}	2^{42}	2^{51}
16	2^5	2^{13}	2^{13}	2^{25}	2^{26}	2^{37}	2^{37}	2^{47}

to further reduce the complexity of error correction. Studies of DRAM errors and their distribution indicate that some error patterns are more likely than others [4, 44, 45]. As discussed in Section 3, the observed errors depend on the underlying fault in the DRAM chips. We can use this knowledge to improve the performance of hash-based correction significantly. Considering the established error patterns reduces the number of possible error distributions greatly. Thus, we extend the analysis of the error correction behavior with regard to the failure modes defined in Section 3. For each failure mode, we give the worst-case number of trial computations. Furthermore, we investigate if parity bits are suited to increase the correction performance even further.

Error correction for F1 failures. Errors of type F1 are the most common errors in modern DRAM devices [45]. Thus, it is essential that their detection is guaranteed and that the correction procedure is efficient. In a purely hash-based design, we cannot guarantee that all single-bit errors are detected. As the location of the flipped bit is unknown, $\mathcal{C}$ is equal to 512, as each bit needs to be flipped and tested. When using additional parity bits, error detection is guaranteed, and the correction efficiency is increased. Given that we add p parity bits, the worst-case number C for F1 failures is $C(F1) = 512/p$. This failure mode strongly profits from a parity-assisted approach as each parity bit linearly reduces the required correction time.

Error correction for F2 failures. Failures of type F2 affect each 64-bit word in DRAM. In each word, exactly one bit at the same offset is stuck at either 0 or 1. All bits are stuck at the same value. Thus, up to 8 bits per cache line are potentially faulty. Table 2 shows that $\mathcal{C}$ and, thus, the correction latency for random 8-bit errors is infeasibly high. Hence, a naïve brute-force approach would not be able to correct a failure of type F2 within reasonable time boundaries.

The correction behavior can be improved by the fact that the *location* of erroneous bits follows a certain pattern for F2 failures. Each pin contributes exactly 1 bit per 64-bit word. Hence, errors that stem from the same stuck pin are at least 64 bits apart. Thus follows that the faulty bits can be located at 64 different positions, depending on which pin of which DRAM chip has failed. With this consideration, we can limit the brute-force search to error masks that follow the failure pattern. Using a pattern-based approach decreases the complexity from $\binom{512}{8}$ to $64 \cdot 2^8 = 2^{14}$ trial computations. This simple modification of the search reduces the worst-case number of trial computations by a factor of 2^{42}.

Parity bits can capture or partially capture F2 failures, depending on the parity granularity and the original bit values. Configurations in which each parity block experiences one erroneous bit due to the stuck pin will detect the failure. Note that an F2 failure does not necessarily produce a parity mismatch. If a bit at the location of the stuck pin is equal to the stuck value, it will not influence the parity computation. Assuming that each bit is equally probable to hold a 1 or 0, we expect that approximately half of all blocks will experience a parity mismatch. Given that there is no other parallel failure mode, the parity bits can help to reduce the number of trial computations if at least 8 parity bits are used. For such configurations, a parity match for a block suggests that the original bit value equals the stuck value. Thus, it is not necessary to include the block when calculating the error mask. Each block that does not experience a parity mismatch will halve the number of trial computations. In the worst case, however, none of the blocks can be skipped.

When using 16 parity bits, the localization of the stuck pin is simplified. As each parity block includes only 32 bits, parity mismatches can be used to determine in which half of the 64-bit word the stuck pin is located. This reduces $\mathcal{C}$ to $32 \cdot 2^8 = 2^{13}$.

We can further improve the performance of the correction. Given that the fault pattern is due to a stuck pin, we know that each bit read from the pin will have the same value. The correction algorithm can search the data for locations where all potentially affected bits are equal. Thus, the algorithm can limit the correction to parts of the cache line that follow the failure pattern. Given that we do not know the content of the cache line, it is possible that benign data also shows a pattern that is equal to the pattern of an F2 failure. We denote such cases as a *pattern collision*. Assuming that the data is uniformly distributed, i.e., each bit has a probability of 0.5 to be either 1 or 0, we can calculate the probability that non-erroneous data causes a pattern collision.

A pattern collision can only occur if 8 bits at the same offset in each beat of the burst are equal. As we consider each bit to be independent of all other bits, the probability that 8 bits are equal is the joint probability of each bit holding a certain value. Hence, the probability that 8 bits are equal is 0.5^8. Assuming a single stuck pin, the probability that no pattern collision occurs is $(1-0.5^8)^{63} \approx 78\%$. Thus, in most cases, the stuck pin will be correctly identified. In such cases, the computational complexity reduces to 2^8 trial computations. Additional parity bits can further decrease $\mathcal{C}$, as they allow the algorithm to ignore blocks without parity mismatches. If one or more pattern collisions occur, the number of trial computations increases as each collision requires 2^8 trial computations to be identified as such.

Clearly, using the available information on fault patterns helps to reduce the number of trial computations by a significant margin. While F2 failures would not be correctable by a naïve brute-force approach, they are easily corrected in the modified approach. Parity bits can further help to refine the localization of the erroneous bits, but their contribution is rather limited.

Error correction for F3 failures. The error correction for the F3 failure mode is very similar to the one for the F2 failure pattern. Even the simplest F3 failure, i.e., 2 stuck pins, generates up to 16 erroneous bits in a cache line. Extrapolating from the results in Table 2, we can assume that a brute-force correction of 16 bits is highly infeasible as the number of

possible 16-bit error masks distributed over a cache line is approximately 2^{100}. However, using the same approach as with the F2 failure mode allows us to correct F3 failures. The number of trial computations for f stuck pins is given by the formula $16\binom{4}{f}2^{8f}$ for x4 chips and by $8\binom{8}{f}2^{8f}$ for x8 chips. For the F3M failure mode, we can give the worst-case number of trial computations for f stuck pins as $\binom{64}{f}2^{8f}$. Compared to F3S, the computational complexity for F3M increases drastically with each additional stuck bit. Parity bits do not necessarily lead to better worst-case behavior as they may not be able to capture the errors. Unlike in the case of F2 failures, blocks without a parity mismatch do not indicate that the block is error-free. A block that suffers two errors will still cause a parity match. Thus, it is not possible to skip blocks without parity mismatches during the correction procedure.

Error correction for F4 failures. Complete chip failures are computationally expensive to correct. For x4 chips, a cache line experiences up to 32 erroneous bits. In the case of x8 chips, this number increases to 64 bits. Thus, the worst-case number of trial computations is $16 \cdot 2^{32}$ and $8 \cdot 2^{64}$, respectively. Unfortunately, it is not possible to decrease the worst-case number of computations for both cases, as all possible patterns have to be tested. Correcting a faulty x4 chip is feasible, albeit the correction latency is non-trivial. In the case of an x8 chip fault, error correction is not possible as the computational complexity is too high.

Error correction for F5 failures. In the combined failure mode, F3 failures are extended by a single, additional transient fault. We assume that the transient fault occurs in a location that is not affected by the stuck pins. Otherwise, the transient fault would never manifest itself as an error. For F3S and F3M errors with f stuck pins, the additional single-bit error means that $\mathcal{C}$ is multiplied by $512 - 8f$.

Adding parity bits does not improve the worst-case number of computations. While the block that experiences the additional error will change its parity, it is not possible to deduce the location of the flipped bit.

From the above results, we can conclude that the additional information on common fault patterns greatly improves the correction capabilities of hash-based integrity protection. Also, it seems that x4 configurations are favorable as the correction complexity does not increase as fast as with x8 devices in the case of F3 and F4 failures. Furthermore, it is possible to correct complete chip failures in x4 configurations.

4.5 Hash Corruption

As the hash is also stored in memory it is, like data, susceptible to corruption. We have to account for this possibility by explicitly checking for a hash corruption. For that, we follow the approach described in [21]. By comparing the stored to the computed hash before initiating error correction, we can determine whether the corruption occured in the hash or the data. If the hamming distance between the two values is low, it is likely that the difference stems from a hash corruption. Contrary to that, a large hamming distance indicates a data corruption.

In the case of a hash corruption, the erroneous hash can be fixed by overwriting it with the hash calculated over the data. Unlike the iterative correction for data errors, this case introduces no additional correction latency. The updated hash value will be stored during the next writeback of the affected cache line.

To identify hash corruption, we have to select the maximum allowed hamming distance d between two hashes. It is reasonable to adjust this distance according to the size of the hash. A larger hash is more likely to experience multiple faults than a small hash. Allowing for d erroneous bits between two k-bit hashes weakens the hash by a factor of $\sum_i^d \binom{k}{i}$ [21]. This is, however, only relevant if the checksum of some erroneous data has a distance of d or less to the correct checksum. Once we decided that an error occurred in data, we do not consider d during the correction procedure. Thus, allowing for hash correction only affects the error detection capability. By balancing d and k, we can offer strong probabilistic error detection while allowing for d-bit errors in the hash.

5 Combining Tagged Memory & Integrity

In this section, we demonstrate that parity-assisted hash-based integrity protection offers the flexibility of implementing memory tagging schemes at near-zero cost. Based on the insights of our analysis in Section 4, we give a concrete instantiation of a scheme that provides integrity protection while allowing for additional bits usable for other purposes. We analyze established tagged memory architectures and show how they can be implemented and combined with a hash-based integrity protection scheme.

5.1 Design of the Integrity Protection

Based on the results of Section 4, we select an integrity protection scheme that uses 8 parity bits in combination with a low-latency hash function for integrity protection. By using 8 parity bits, we ensure that all F1 failures are detected and efficiently corrected. We do not limit our design to a fixed hash size. Instead, we select the hash size according to the requirements of the tagging scheme that we want to implement. Tagging schemes that permit a large hash size allow stronger reliability and correction capabilities when compared to schemes with smaller hashes.

Protecting tag bits. When adding memory tags, we must ensure that bit errors in the tag are also detected. How we include the tags in the hash depends on the structure of the hash function. When using the structure given in Figure 1, we distribute the tag bits over the tweaks of the `QARMA` instances. For the construct shown in Figure 3, we include the tag bits

Table 5: Table showing how a hash-based integrity protection scheme can be instantiated for existing tagged memory architectures and which security guarantees, in terms of error correctability, it provides.

Architecture	Tag Size	Granularity	Bit Distribution			Faulty Bits in Hash (d)	Reliability	Correctable Failure Modes							
			Parity	Hash	Tag			F1	F2	F3S(8,4)	F3S(4,4)	F3M	F5S(4,4)	F5S(8,_)	F5M
DIFT [47], HDFI [42] Shakti-t [33], M-Machine [6]	1 bit	8 bytes	8	+ 48	+ 8	4	2^{33}	✓	✓	✓	✓	3	✓	(8,3)	2
SPARC ADI [1]	4 bits	64 bytes	8	+ 52	+ 4	5	2^{34}	✓	✓	✓	✓	3	✓	(8,3)	3
CHERI 128 [53,55]	1 bit	16 bytes	8	+ 52	+ 4	5	2^{34}	✓	✓	✓	✓	3	✓	(8,3)	3
CHERI 256 [53,55]	1 bit	32 bytes	8	+ 54	+ 2	5	2^{36}	✓	✓	✓	✓	3	✓	(8,4)	3
ARM MTE [26]	4 bits	16 bytes	8	+ 40	+ 16	4	2^{25}	✓	✓	(8,3)	✓	2	(4,2)	(8,2)	2
lowRISC [30]	4 bits	8 bytes	8	+ 24	+ 32	2	2^{15}	✓	✓	✗	✗	✗	✗	✗	✗
Model A	32 bits	64 bytes	1	+31	+32	3	2^{19}	✓	✓	(8,2)	(4,2)	2	✗	✗	✗
Model B	46 bits	64 bytes	1	+17	+46	1	2^{11}	✓	✓	✗	✗	✗	✗	✗	✗
Model C	51 bits	64 bytes	1	+12	+51	1	2^{6}	✓	✗	✗	✗	✗	✗	✗	✗
No Tagging, SEC-DED	-	-	64	-	-	-	1	✓	✓	✗	✗	✗	✗	✗	✗
No Tagging, Chipkill	-	-	64	-	-	-	1	✓	✓	(✓)	✓	✗	✗	✗	✗

in the additional data bits. Thus, the hash protects not only the data, but the tag as well. Detecting and correcting errors in the tag bits works the same as with data errors. On a hash mismatch, we can incrementally apply error masks on the tag bits and check the resulting hash. As the tag space is small, correcting errors in the tag bits is far more efficient than data correction. The correction latency depends on the actual size of the tag space and, thus, on the implemented tagged memory architecture. Given that we have, at most, 63 bits for the tag, the expected number of iterations is small compared to the results provided in Section 4.3.

Figure 4 shows the principle design of our approach. The memory controller is augmented with an instance of the hashing function. On write requests, the checksum over the data and the corresponding tags is computed. This checksum is then stored next data and tag in the chips usually reserved for ECC. When reading from memory, the checksum is recomputed and compared to the stored checksum. If both values match, the data is forwarded to the CPU. Depending on the implemented tagged memory architecture, the tag may be forwarded or compared to a reference tag directly in the memory controller. On a checksum mismatch, the memory controller generates a machine check exception (MCE) and commences the correction procedure.

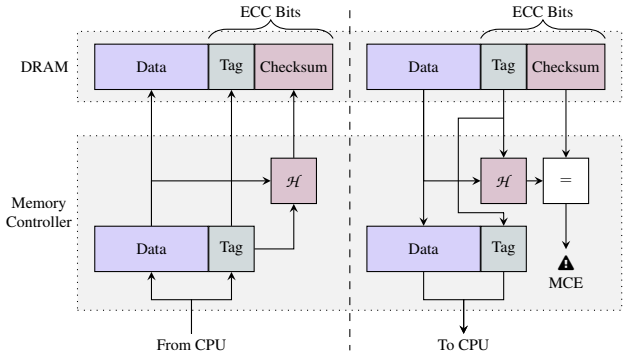

Figure 4: A high-level overview of hash-based integrity protection with tagging. When writing to memory (left), a checksum is computed over the provided tag and the data, and stored in DRAM. On read accesses (right), the stored checksum is compared with a freshly computed checksum, thus detecting errors in the data and tag bits. This initiates error correction.

5.2 Tagged Memory Applications

Various tagged architectures [19] introduce fine-grained security policies tailored toward different threat models and applications. These architectures require memory tagging with different tag sizes and different granularities. As our design is flexible regarding the hash size, implementing most of the tagged memory schemes is feasible. Table 5 analyzes how our protection scheme can be instantiated to facilitate the different requirements of existing tagged memory architectures. It shows that schemes are feasible to implement using the available redundancy bits without losing error correction or detection capabilities. The bit distribution is derived from the utilized tag size and granularity of the implemented tagged architecture. We compute the reliability as the ratio between the rate of undetectable faults in commodity codes and the expected rate of undetected faults in our scheme. This yields the formula

$$\frac{R_{UD}}{R_{UD,H}} = \frac{R_{UD}}{R \cdot 2^{-k}} \tag{6}$$

for a k-bit hash. Essentially, the reliability is the factor by which the rate of undetectable faults (R_{UD}) is decreased. A value of 1 would mean an equal reliability as commodity solutions using linear codes. As we need to account for d errors in the hash, we adjust the listed reliability according to Section 4.5. This only affects the initial error detection step but has no effect on the error correction. We select d according to the size of the hash. For larger hashes, we allow for more erroneous bits, while smaller hashes may not differ in as many bits. We give the next-lower power-of-two instead of the exact reliability value to allow for a more intuitive representation. For each possible configuration, we show which of the failure modes defined in Section 3 can be corrected. We only consider a failure mode to be correctable if the number of trial computations does not exceed the unmodified reliability. We

do not list F4 as a separate failure mode. F4 correction is only possible if the scheme can correct F3S failures where all pins are stuck. As a reference, we list the correctable failure modes for commodity SEC-DED and Chipkill codes that utilize 64 parity bits.

In the following, we highlight the bit distribution for different schemes, divided into parity, hash, and tag bits. Our design suggests using 8 parity bits (cf. Section 4) in combination with the remaining hash bits for error detection and correction.

Architectures utilizing a single-bit tag on a word granularity, like DIFT [47], HDFI [42], Shakti-t [33], and the M-Machine [6], reserve 8 bits per cache line for the tag value. The remaining bits are used for parity and the hash, yielding 8 and 48 bits, respectively. Similarly, CHERI [53, 55] also utilizes a single-bit tag. Depending on the configuration, however, the granularity of CHERI is either 16 or 32 bytes. The 16-byte capability model utilizes 4 bits per cache line, while the 32-byte model requires only 2 bits. These schemes allow for 52 and 54 hash bits, respectively. These large hashes mean that almost all failure modes are correctable.

Multi-bit schemes like ARM MTE [26], SPARC ADI [1], and lowRISC [30] require a larger tag size depending on the tag granularity. Thus, they do not allow for as much reliability and error correction as schemes with smaller tags. Precisely, ARM MTE and SPARC ADI utilize a 4-bit memory tag per 16 and 64 bytes, respectively. SPARC ADI only requires 4 bits per cache line to store the tag information, leaving 52 bits for the hash. In contrast, ARM MTE requires 16 bits per cache line resulting in a 40-bit hash. The lowRISC tagged memory architecture requires 32-bit memory tag per cache line, resulting in a 24 bit hash. Relatedly, the SPEAR-V enclave uses 24-bit tags per page [38]. When stored in each cache line instead, it would allow for a 32-bit hash with 8 parity bits. For these schemes, while the most basic error correction is still possible, complex failure modes are not correctable due to the large number of trial computations they require.

We model three additional tagged memory schemes (Model A/B/C) to show the possibilities and limits of hash-based integrity protection. To maximize the available tag bits, we reduce the number of parity bits down to the smallest possible value (1) that still guarantees single-bit error detection. This allows storing up to 51 additional tag bits. Model A offers the same number of tag bits per cache line as lowRISC [30]. However, due to the larger hash size, the range of correctable failure modes is increased. Reducing the number of parity bits, however, increases the error correction latency for F1 and F2 failures. Next, we define Model B, allowing 46 tag bits per cache line. Due to the small hash size, only F1 and F2 failures can be corrected. Finally, Model C yields the maximum number of possible tag bits (51) while still guaranteeing single-bit error detection. While this model does not allow for complex error correction, correcting single-bit errors is still possible.

Overall, we find that the hash-based approach is highly feasible. Depending on the size of the hash, the correctable failure modes differ. However, the detection ability outperforms both SEC-DED and Chipkill, given that the hash size is selected accordingly.

6 Evaluation

The main advantage of co-locating memory tags and hashes is that memory tags do not need to be loaded via additional fetch operations. Our design completely eliminates the need for additional tag fetches. However, computing the hash introduces additional latencies in memory reads and writes.

In a naïve tagged memory implementation that caches tags in combination with the associated data, each cache miss will result in an additional DRAM memory access. With our proposed design, the additional latency that is added to every memory read and write depends on the specific hash implementation and its latency. We investigate the performance overhead for four different configurations using the hash functions described in Section 4.1.

6.1 Implementation in gem5

We base our evaluation on the gem5 simulator [29]. We implement a tagged memory architecture that resembles ARM MTE [26] and caches tags in parallel to the respective data. When accessing DRAM, we perform an additional read access that fetches the associated tag from a reserved memory region. Our model uses 4-bit tags for each 16 bytes of data. Accesses to the dedicated tag memory region are always performed with cacheline granularity.

Our novel scheme is simulated by introducing an additional hashing module in the memory controller. This module stalls every memory access as specified by the latency of the used hash function and computes the hash over the data in the memory transaction. The hash latency is imposed on downstream write requests and upstream read responses. Read requests that are directly served by the memory controller's write queue do not experience additional latencies as no additional DRAM fetch is performed. Once a memory packet reaches its ready time, it is forwarded to the next component in the memory hierarchy. Our changes are transparent to the operating system and require no software changes or modifications outside of the memory controller. Table 6 lists the gem5 configuration that is used when measuring the performance overheads. Due to limitations of the simulator, the main memory is split into one 3GB and a second 8GB module. Each module resides in its own channel. For the tagged memory architecture, we reserve a fraction of the DRAM as a tag storage. We do the same for our scheme to allow for a fair comparison, as differing DRAM capacities could lead to performance impacts.

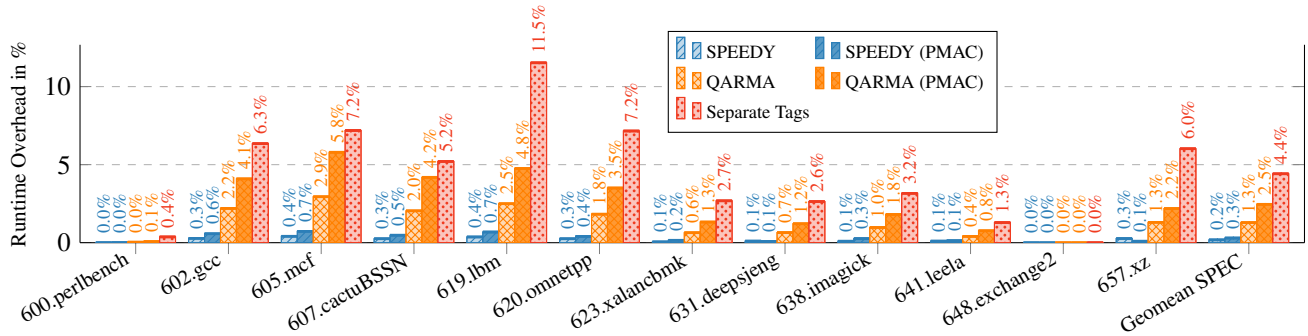

Figure 5: Simulated performance overhead using the SPEC CPU2017 benchmark suite.

Table 6: The gem5 configuration used in our evaluation.

Core	KVM CPU / Timing CPU at 3 GHz			
L1D Cache	64 kB, 8-way, 1-cycle latency			
L1I Cache	16 kB, 8-way, 1-cycle latency			
L2 Cache	256 kB, 4-way, 10-cycle latency			
DRAM	3GB + 8GB, DDR4-2400			
Hash Latency	SPEEDY 0.3 ns	QARMA 2.2 ns	SPEEDY-PMAC 0.6 ns	QARMA-PMAC 4.4 ns
Kernel	Linux 5.15.67			

For our scheme, we evaluate four latency settings: SPEEDY with a latency of 0.3 ns, QARMA with 2.2 ns additional latency, SPEEDY-PMAC with a latency of 0.6 ns and QARMA-PMAC with a latency of 4.4 ns.

Measuring performance overheads. We run the SPEC CPU 2017 benchmarks in three different system configurations. An unmodified simulator configuration is used to establish baseline performance values. The configuration that implements the tagged memory architecture allows us to estimate the performance overhead due to additional tag fetches. Finally, the configuration implementing our scheme showcases the overhead reduction. As described, we measure the performance impact of our scheme for four different hash instances. SPEEDY and QARMA refer to the implementation as shown in Figure 1. The PMAC variants refer to their cryptographically secure counterparts as described in Section 4.1. Some of the benchmarks caused compile errors or would not run in the simulator and were, thus, not executed.

When booting the system, we use the KVM CPU to create a checkpoint and reduce the overall simulation time. Upon reaching the start of the benchmark run, we switch to the accurate CPU model that implements the tagged memory architecture and our scheme. As our main performance metric, we measure the number of CPU cycles of each benchmark. We execute each benchmark multiple times and average over the results to get a stable result as well as the standard deviation. Due to the precision of the simulator, we find that individual benchmark runs for a fixed configuration show a negligible spread in CPU cycles with a standard deviation that is two to three orders of magnitude lower than the recorded cycle count.

Figure 5 shows the relative performance overheads for all SPEC CPU 2017 benchmarks executable in our simulator. For all of the benchmarks, our hash-based approach outperforms the tagged architecture. However, the combination of QARMA and the PMAC design tends to impose non-trivial latencies, sometimes causing overheads almost equal to the original tagging implementation. This result underlines the importance of using a hashing function with minimal latency, as stalling memory packets has a strong impact on system performance for traffic-intensive workloads. Interestingly, SPEEDY-PMAC performs better than SPEEDY for 657.xz. We find that the additional latency causes write requests to stay in the write queue of the memory controller for a longer duration. This, in turn, increases the number of read requests that are directly serviced by the write queue and do not trigger a DRAM fetch. Thus, in rare cases, stalling each write request for a short amount of time can improve the read performance.

6.2 Correction Performance

We evaluate the error correction performance for two settings. In the first setting, we look at arbitrarily distributed f-bit errors. These errors are corrected using a brute-force approach. The second setting discusses the error correction using patterns as described in Section 4.4.

Brute-force correction. When relying on a brute-force approach with additional parity bits, the correction time increases with each additional faulty bit. As we do not know the number of faulty bits beforehand, we must rely on an incremental approach. This means that we increment the number of assumed faulty bits after exhausting the search space. Figure 6 illustrates the correction time when using the SPEEDY cipher and 4, 8 and 16 parity bits, respectively. The correc-

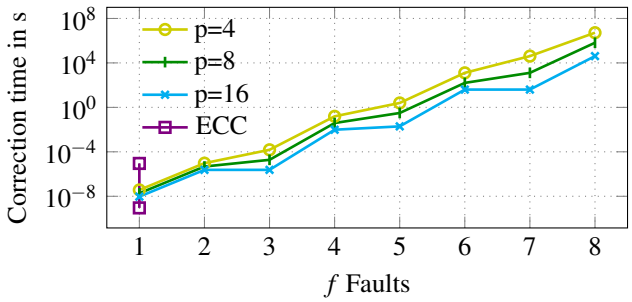

Figure 6: The worst-case brute-force error correction time in dependency of the number of corrected bits.

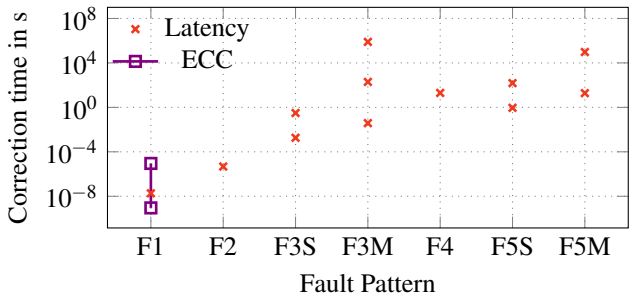

Figure 7: The worst-case correction time for different error patterns for a configuration with x4 devices and 8 parity bits.

tion of single-bit errors, the most common type of fault, takes between 9.6 ns and 38.4 ns, depending on the number of parity bits. Kwong et al. find that in commodity ECC devices, an access that triggers a correction shows a latency that is 5 orders of magnitude slower than a regular access [23]. For an average DRAM access time of 90 ns [48], this translates to a correction latency of approximately 9 ms. However, Cojocar et al. [9] find a correction overhead of 1% for certain configurations. Thus, we assume a minimum correction latency of 0.9 ns and a maximum latency of 9 ms for commodity ECC. Clearly, the applicability of a brute-force approach is limited by the available time budget. Due to the flexibility of our design, it is possible to set a fixed time margin after which an error is considered uncorrectable. While it would be possible to correct an 8-bit error in approxmiately 11 h, it is much more feasible to consider fault patterns for correction.

Pattern-based correction. Correction times drastically improve when considering the error patterns described in Section 4.4. Figure 7 shows the correction times for a configuration with 8 parity bits and x4 DRAM chips. We give the correction latencies for up to three stuck pins for F3S, F3M, F5S, and F5M. F2 faults cause up to 8 faulty bits in a cache line. Our scheme corrects 8-bit F2 faults in 4.75 µs. Compared to the brute-force approach, in which 8-bit faults are not correctable within a reasonable time, this is a considerable improvement. For F3S and F3M, the correction time strongly depends on the number of stuck pins. While two stuck pins (16 faulty bits) can be corrected in 1.84 ms, correcting three stuck pins (24 faulty bits) already takes 311.4 ms for F3S. Due to the larger search space for F3M, double-pin correction takes 38.31 ms and triple-pin correction takes over three minutes. A F4 fault, i.e. a complete fault of an x4 chip, can be corrected within 19.93 s. For F5S and F5M, the correction time only depends on the underlying F3 fault type.

6.3 Hardware Overhead Estimation

In our design we propose adding multiple block cipher instances. For optimal correction performance we use three `SPEEDY` instances (cf. Figure 3) per DRAM channel to calculate the hash for a cache-line at once. In the case of `QARMA`, 8 instances are required. This would require 104 or 119 kGE for fully unrolled instances of `SPEEDY` and `QARMA`, respectively [24] when using a 15 nm process . Avanzi [2] reports 22.6 kGE (1238.1 µm^2) for a latency optimized `QARMA` using a 7 nm process. Thus, for 8 `QARMA` instances, we would require 181 kGE (9905 µm^2). This is approximately 0.0038% of a Raptor Lake CPU with a comparable feature size and an area of 257 mm^2 [49]. This overhead can be optimized in two ways. First, by using smaller (e.g., round-reduced) ciphers (cf. Section 4.1). And second, when deprioritizing correction performance, the number of block cipher instances can be reduced to 1 or 2 for `SPEEDY` and `QARMA`, respectively. This has no impact on performance when there are no errors, since they can still saturate one full DRAM channel.

Leander et al. only provide estimated power requirements for operation at 100 MHz [24]. Simulating power consumption for a more realistic frequency of 3 GHz requires commercial EDA tools with accurate power models. Thus, similar to related work, we cannot provide concrete numbers for estimating the power consumption.

7 Discussion

This section discusses related work and possible future work.

7.1 Related Work

The combination of memory tags and DRAM integrity protection codes is a well-studied topic. Gumpertz et al. [15] propose a method to integrate tags into ECC codes that slightly weakens the error correction and detection capabilities of ECC. In their approach, they do not actually store the tag bits in memory. Instead, the tag has to be presented in a register on each access. Their scheme implicitly encodes the tag into the redundancy information. Thus, accessing data with an incorrect tag will cause an error that is encoded similarly to a double-bit error. While this approach allows for low-overhead storage of tags, it has several drawbacks. Double-bit errors that appear in combination with tag mismatches will either

be undetected or even miscorrected. Moving data to a different physical location without knowing the tag is impossible. As the tag is never explicitly stored next to the data, using multiple tags requires complex tag management.

MUSE [31], a novel variant of multi-use ECC codes, proposes to use a multiplicative encoding that detects errors as invalid remainders during decoding. Their codes allow them to encode additional tags into the redundancy information such that the tags can be recovered when reading from memory as they are stored as part of the remainder. For a 64-bit word, their code provides single-bit correction and probabilistic double-bit detection. Contrary to commodity SEC-DED codes, MUSE cannot detect all double-bit errors.

Our hash-based approach differs significantly from the above. Storing the tag in memory allows copying and relocating tagged data without complex tag management. As the tag is completely decoupled from the integrity verification, it is possible to perform the hash validation while forwarding the data to the CPU. This feature is similar to asynchronous reporting in ARM MTE [26]. While the error detection of a hash-based approach is probabilistic, it performs much better than what is achievable by linear codes. Linear codes are inherently limited by their hamming distance, while hash-based integrity protection is not. Especially for fault patterns that combine transient and hard faults, reliable error detection is hard to achieve using linear codes.

Protection of DRAM integrity through cryptographic building blocks is a relatively novel approach. It originally stems from the need for better integrity protection in the wake of Rowhammer attacks that induce bit flips in DRAM [22]. Cojocar et al. [9] show that the error detection capabilities of linear codes are not fit to detect bit flips that are induced on purpose. Thus, functions with strong diffusion, like MACs or hashes, are more suitable for protecting DRAM integrity.

Saileshwar et al. [37] propose Synergy, a design that repurposes the additional ECC memory chips to hold a message authentication code. The main goal of their approach is to use MACs of integrity trees as a means of detecting DRAM integrity violations. Synergy is based on the observation that, in the common case, no such integrity violation occurs. An additional chip holds parity data that, in the case of an integrity violation, is accessed to correct the faulty data. While Synergy is able to correct a single chip fault, it has to do so in an iterative approach. As Synergy uses the MACs from an integrity tree and additional parity, it introduces an additional 12.5% memory overhead.

Qureshi et al. [35] propose to increase the granularity over which a checksum is computed from 64 bits to 512 bits. Furthermore, they use a MAC to protect data integrity. Building upon the idea of replacing linear codes with MACs, they then introduce integrity-protected ECC memory (IPEM) and integrity-protected Chipkill memory (IPCM). In IPEM, each cache line is protected by a 56-bit MAC and a 10-bit linear code offering SEC. The MAC detects arbitrary bit faults with high probability. The linear code corrects single-bit faults efficiently. While commodity SEC-DED can correct up to 8 single-bit errors per cache line, IPEM can only correct a single-bit fault. The authors argue that multi-bit faults in multiple words of a cache line occur with negligible probability. Unfortunately, IPEM cannot protect against errors due to a single stuck pin or large-granularity faults. To solve this issue, the authors propose IPCM. In IPCM, the two chips that hold parity data in Chipkill codes store a MAC and additional parity information. The error correction in IPCM follows an iterative approach, as the used code does not allow to deduce the location of the failed chip.

Juffinger et al. [21] and Fakhrzadehgan et al. [13] both propose to mitigate Rowhammer through the use of MACs instead of linear codes. Their schemes detect errors that would bypass the error detection in the case of linear codes. As they reuse the existing ECC memory, the storage overhead is equal to the 12.5% found in commodity ECC DRAM. Like other hash- or MAC-based approaches, the error correction is implemented as an iterative approach.

While related research explores error detection and correction through MACs, they do not explore the possibility of co-locating metadata by reducing the output size. We eliminate this blind spot and show that hash-based integrity protection is not only feasible but allows us to efficiently integrate memory tags into ECC DRAM, thus eliminating the need for explicit tag fetches.

7.2 Future Work

With the introduction of DDR5 memory, the possibility for future hash-based schemes is manifold. In DDR5 ECC DRAM, the available storage for ECC bits is doubled to 25%. Thus, we can further increase the size of the hash or add more parity bits to the design. Naturally, a larger storage space for ECC bits allows for more powerful linear codes than the ones that are used in commodity DDR4 ECC DRAM modules. Most DDR5 modules implement on-die ECC that corrects single-bit errors. As the on-die ECC does not support double-error detection, it will miscorrect errors that exceed one bit per codeword. Thus, future error correction designs have to account for the potentially increased number of errors due to miscorrection. We expect failure modes that include stuck pins to become even more important in this new DRAM generation. As DDR5 increases the burst length while decreasing the size of the single beats, each stuck pin will potentially cause double the number of errors per cache line.

8 Conclusion

In this work, we analyzed the error detection and correction capabilities achievable with hash-based designs under specific fault models motivated by real-world data. Based on our results, we show that a variety of tagged memory architectures

can be implemented by utilizing free bits in ECC DRAM modules. Co-locating bits for memory tagging and hashes for integrity protection is a practical optimization to eliminate the bandwidth overheads of tagged architectures. In contrast to the integration of tags into linear codes [14, 15, 31], the combination of hash-based integrity protection and memory tagging is a novel approach that has not been demonstrated in a concrete scheme. Our approach allows us to utilize up to 16 tag bits on a granularity of 64 bytes while providing detection and correction capabilities that exceed the capabilities of commodity linear codes. This tag size allows for the implementation of promising tagged architectures like CHERI, ARM MTE, and SPARC ADI, while drastically reducing overheads due to tag fetches. Our evaluation shows that, for an MTE-like architecture, we can reduce performance overheads by an average factor of 20. Furthermore, we highlighted evaluation models for tagged architectures based on large tag sizes (32, 46, and 51 bits per cache line) to show the limitations of our approach and to facilitate possible future work.

9 Acknowledgements

We thank the anonymous reviewers and our shepherd for their valuable feedback and suggested improvements. This project has received funding from the Austrian Research Promotion Agency (FFG) via the SEIZE project (FFG grant number 888087). Additional funding was provided by generous gifts from Intel and from SGS.

References

[1] Aingaran et al. M7: Oracle's Next-Generation Sparc Processor. *IEEE Micro*, 2015.

[2] Roberto Avanzi. The QARMA Block Cipher Family. *IACR Trans. Symmetric Cryptol.*, 2017.

[3] Robert Baumann. Soft Errors in Advanced Computer Systems. *IEEE Des. Test Comput.*, 2005.

[4] Bautista-Gomez et al. Unprotected computing: a large-scale study of DRAM raw error rate on a supercomputer. In *SC'16*, 2016.

[5] Buchanan et al. When good instructions go bad: generalizing return-oriented programming to RISC. In *CCS'08*, 2008.

[6] Carter et al. Hardware Support for Fast Capability-based Addressing. In *ASPLOS'94*, 1994.

[7] Checkoway et al. Return-oriented programming without returns. In *CCS'10*, 2010.

[8] Tony Chen and David Chisnall. Pointer tagging for memory safety, 2019.

[9] Cojocar et al. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *S&P'19*, 2019.

[10] Crandall et al. Minos: Architectural support for protecting control data. *ACM Trans. Archit. Code Optim.*, 2006.

[11] Dalton et al. Raksha: a flexible information flow architecture for software security. In *ISCA'07*, 2007.

[12] Timothy J Dell. A white paper on the benefits of chipkill-correct ecc for pc server main memory. *IBM Microelectronics division*, 11(1-23):5–7, 1997.

[13] Fakhrzadehgan et al. SafeGuard: Reducing the Security Risk from Row-Hammer via Low-Cost Integrity Protection. In *HPCA'22*, 2022.

[14] Richard H. Gumpertz. *Error Detection with Memory Tags*. PhD thesis, 1981.

[15] Richard H. Gumpertz. Combining Tags With Error Codes. In *ISCA'83*, 1983.

[16] Sudhanva Gurumurthi. Advanced memory device correction (amdc) for servers. *AMD Whitepaper*, 2020.

[17] Ruirui C. Huang and G. Edward Suh. IVEC: off-chip memory integrity protection for both security and reliability. In *ISCA'10*, 2010.

[18] JEDEC Standard. Low Power Double Data Rate 4. *LPDDR4, JESD209-4A,(Revision of JESD209-4)*, 2014.

[19] Jero et al. TAG: Tagged Architecture Guide. *ACM Comput. Surv.*, 2023.

[20] Joannou et al. Efficient Tagged Memory. In *ICCD'17*, 2017.

[21] Juffinger et al. CSI: Rowhammer–Cryptographic Security and Integrity against Rowhammer. *S&P'23*, 2022.

[22] Kim et al. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA'14*, 2014.

[23] Kwong et al. RAMBleed: Reading Bits in Memory Without Accessing Them. In *S&P'20*, 2020.

[24] Leander et al. The SPEEDY Family of Block Ciphers Engineering an Ultra Low-Latency Cipher from Gate Level for Secure Processor Architectures. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2021.

[25] Levy et al. Lessons learned from memory errors observed over the lifetime of Cielo. In *SC'18*, 2018.

[26] Arm Limited. Memory tagging extension: Enhancing memory safety through architecture, 2019.

[27] Liu et al. An experimental study of data retention behavior in modern DRAM devices: implications for retention time profiling mechanisms. In *ISCA'13*, 2013.

[28] Liu et al. TMDFI: Tagged Memory Assisted for Fine-Grained Data-Flow Integrity Towards Embedded Systems Against Software Exploitation. In *TrustCom'18*, 2018.

[29] Lowe-Power et al. The gem5 Simulator: Version 20.0+. *CoRR*, 2020.

[30] lowRISC Team. Tag support in the rocket core. https://lowrisc.org/docs/minion-v0.4/tag_core/, 2017. Accessed: 2022-10-01.

[31] Manzhosov et al. MUSE: Multi-Use Error Correcting Codes. *CoRR*, 2021.

[32] Marazzi et al. ProTRR: Principled yet Optimal In-DRAM Target Row Refresh. In *S&P'22*, 2022.

[33] Menon et al. Shakti-T: A RISC-V Processor with Light Weight Security Extensions. In *HASP'17*, 2017.

[34] The Chromium Project. Memory Safety. https://www.chromium.org/Home/chromium-security/memory-safety/. Accessed: 2022-10-01.

[35] Moinuddin Qureshi. Rethinking ecc in the era of rowhammer. In *First workshop on DRAM Security, co-located with ISCA*, 2021.

[36] Phillip Rogaway. Efficient Instantiations of Tweakable Blockciphers and Refinements to Modes OCB and PMAC. In *ASIACRYPT'04*, 2004.

[37] Saileshwar et al. SYNERGY: Rethinking Secure-Memory Design for Error-Correcting Memories. In *HPCA'18*, 2018.

[38] Schrammel et al. SPEAR-V: Secure and Practical Enclave Architecture for RISC-V. In *ASIACCS'23*, 2023.

[39] Schroeder et al. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS'09*, 2009.

[40] Serebryany et al. Memory Tagging and how it improves C/C++ memory safety. *CoRR*, 2018.

[41] Hovav Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *CCS'07*, 2007.

[42] Song et al. HDFI: Hardware-Assisted Data-Flow Isolation. In *S&P'16*, 2016.

[43] Sridharan et al. Feng shui of supercomputer memory: positional effects in DRAM and SRAM faults. In *SC'13*, 2013.

[44] Sridharan et al. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *ASPLOS'15*, 2015.

[45] Vilas Sridharan and Dean Liberty. A study of DRAM failures in the field. In *SC'12*, 2012.

[46] JEDEC Standard. JEDEC Standard: DDR4 SDRAM. *JESD79-4, Sep*, 2012.

[47] Suh et al. Secure program execution via dynamic information flow tracking. In *ASPLOS'04*, 2004.

[48] Suzuki et al. Approaching DRAM performance by using microsecond-latency flash memory for small-sized random read accesses: a new access method and its graph applications. *Proc. VLDB Endow.*, 2021.

[49] TechPowerUp. Intel "Raptor Lake" Core i9-13900 Delidded, Reveals a 23% Larger Die than Alder Lake. https://www.techpowerup.com/297506/, 2022. Accessed: 2022-10-01.

[50] Gavin Thomas. A proactive approach to more secure code. *Blog Post*, 2019.

[51] Venkataramani et al. FlexiTaint: A programmable accelerator for dynamic taint propagation. In *HPCA'08*, 2008.

[52] David A. Wagner. A Generalized Birthday Problem. In *CRYPTO'02*, 2002.

[53] Watson et al. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *S&P'15*, 2015.

[54] Witchel et al. Mondrian memory protection. In *ASPLOS'02*, 2002.

[55] Woodruff et al. The CHERI capability model: Revisiting RISC in an age of risk. In *ISCA'14*, 2014.

[56] Yaglikçi et al. BlockHammer: Preventing RowHammer at Low Cost by Blacklisting Rapidly-Accessed DRAM Rows. In *HPCA'21*, 2021.

[57] Yaglikçi et al. Security Analysis of the Silver Bullet Technique for RowHammer Prevention. *CoRR*, 2021.

[58] Zeldovich et al. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *OSDI'08*, 2008.