Voodoo: Memory Tagging, Authenticated Encryption, and Error Correction through MAGIC

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

https://www.usenix.org/conference/usenixsecurity24/presentation/lamster

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

Voodoo: Memory Tagging, Authenticated Encryption, and Error Correction through MAGIC

Lukas Lamster
Graz University of Technology

Martin Unterguggenberger
Graz University of Technology

David Schrammel
Graz University of Technology

Stefan Mangard
Graz University of Technology

Abstract

Confidentiality, authenticity, integrity of data, and runtime security are ubiquitous concerns in modern computer systems. However, these security concerns have traditionally been addressed by separate mechanisms. Error-correcting codes (ECC) detect and correct DRAM errors, ensuring the integrity of stored data. Authenticated memory encryption provides data confidentiality and authenticity. Memory tagging enforces memory safety, thereby improving runtime security. The lack of a combined primitive increases system complexity, memory overheads, and the overall performance impact. In this work, we present Voodoo, the first combined scheme for authenticated encryption, DRAM error correction, and memory tagging. Our design extends the MAGIC mode for authenticated encryption and error correction proposed by Kounavis et al. [31]. With Voodoo, DRAM data is encrypted, and a tag-dependent message authentication code protects the integrity of the stored data while simultaneously allowing for the correction of DRAM faults. Thus, we can implement a wide range of tagged memory architectures *without introducing additional memory requests or storage overheads*. We present three tag encoding schemes providing up to 36 tag bits per cache line. Using the gem5 simulator, we implement and benchmark our design. Our evaluation shows a low runtime overhead of 1.4% on average compared to a system without any of the provided security features. We use a Monte-Carlo simulation of a DRAM fault model based on real-world DRAM fault behavior to demonstrate the corrective capabilities of Voodoo. Our results show that we consistently outperform traditional single-error correction, double-error detection (SEC-DED) codes in terms of error correction and detection. For multi-chip faults, Voodoo offers stronger error detection than commodity Chipkill solutions.

1 Introduction

Security is of utmost importance in modern computing systems. Data confidentiality, authenticity, and integrity are essential, especially in server and cloud systems where multiple mutually distrusted parties share the same hardware. However, only protecting data in DRAM is not sufficient. Memory safety vulnerabilities constantly threaten system security, as security-critical software is predominantly written in memory-unsafe C/C++ code. Traditionally, the security of systems is increased by layering security primitives over each other. This approach may work at the moment, but it ultimately increases system complexity, and the overheads introduced by each new layer accumulate.

At the DRAM level, server-grade systems implement integrity protection. In DRAM chips, memory cells store information as an electrical charge. These cells suffer from leakage currents, causing unwanted data alterations. External disturbances, like radiation or high temperatures, can cause large-scale failures that corrupt multiple bits. Without integrity protection, DRAM errors would cause silent data corruption and, subsequently, erroneous computations or system crashes. Thus, *error-correcting codes* (ECC) are used to detect and correct DRAM errors. An ECC computes and stores a checksum of the stored data. When reading from memory, the checksum can restore the original data if an error is detected. Commodity single-error correction, double-error detection (SEC-DED) codes, for example, can correct one erroneous bit per 64 bits in a cache line. Thus, ECC provides *integrity* for DRAM data.

On top of integrity, keeping data secret and protecting against deliberate data modification is necessary. Memory encryption technologies of major hardware vendors like Intel [22] or AMD [2, 29] provide data confidentiality through encryption. Encryption protects against attacks like bus snooping or cold boot attacks [17, 18]. Moreover, memory encryption provides cryptographic isolation for virtual machines in cloud environments. Encrypting the data of co-located tenants isolates them from each other and protects against a potentially malicious hypervisor. Intel's trust domain technology, Intel TDX [11], provides *authenticated encryption* by extending the encryption with a message authentication code (MAC) based on SHA-3 [15]. A MAC generates a fingerprint of the input data using a secret key. Data modifications are detected by recomputing and comparing the MAC to a stored version.

Only a party possessing the correct key can compute a MAC for a given input. Thus, the MAC provides authenticity and integrity of DRAM data.

While authenticated encryption is a crucial component for safeguarding DRAM data, runtime safety is equally imperative. As data is processed unencrypted at runtime, additional measures must be taken to protect it from vulnerabilities and threats. Studies by Microsoft found that the most prominent sources of security-relevant bugs in their products are memory safety issues [45]. Memory safety violations allow attackers to read or modify allocated data, tamper with heap metadata, or perform other security-critical operations [6, 9, 20, 23, 56]. *Memory tagging* mitigates memory safety issues by associating each allocation with a *tag*. A tag holds metadata for the allocation and is used to enforce a security policy that protects against memory safety violations at runtime. ARM and Oracle provide commercial ISA extensions using tagged memory. ARM MTE [50] and SPARC ADI [51] implement hardware-supported tagging for memory protection. While tagged memory architectures offer strong protection against software-based attackers, they introduce performance and memory overheads since tags must be fetched frequently. Furthermore, tags must be stored in DRAM, thus reducing the amount of usable memory.

Recently, the idea of combined primitives offering a more holistic protection is gaining traction. MACs and ECC, for example, are conceptually similar as they both detect data modifications. However, combining the strong detection capabilities of MACs with the error correction of ECCs is challenging. In 2020, Kounavis et al. proposed `MAGIC` [31], a combined mode for authenticated encryption and error correction. The `MAGIC` mode is a generic construct facilitating authenticated memory encryption while allowing the correction of errors that affect a bounded number of bits in a single granule using a specially tailored MAC function.

In this work, we present `Voodoo`, the first combined scheme for DRAM error detection and correction, authenticated encryption, and memory tagging. Building on `MAGIC`, we develop three novel tag encodings that allow us to seamlessly combine memory tagging with authenticated encryption and error correction. Thus, `Voodoo` overcomes the limitations of existing tagged memory architectures, which incur non-negligible performance and memory overheads. *We can offer up to 36-bit tags without introducing additional memory requests or storage overheads*. We implement our design using gem5 [7] and evaluate its performance. Our results underline the practicability of our design as we reach a performance overhead of 1.4% on average compared to a system without any additional protection. Furthermore, we implement a Monte-Carlo simulation of DRAM errors and benchmark the correction capabilities of our design. *We find that we can reach an error correction rate of* 99% *and an error detection rate of* 100% *for errors confined to a single DRAM chip.* This is a significant improvement compared to SEC-DED, which has a correction rate of approximately 62%.

Contributions. We make the following key contributions:

1. We present `Voodoo`, the first combined primitive for authenticated encryption, DRAM error correction, and memory tagging.

2. We develop three tag encoding schemes, providing up to 36 tag bits while facilitating reliable error detection and correction.

3. We implement a prototype of our design and evaluate the performance using the SPEC CPU 2017 benchmark suite, yielding a geomean overhead of just 1.4%

4. We implement a Monte-Carlo simulation of a DRAM fault model. `Voodoo` reaches a 99% error correction rate for single-chip errors, significantly improving on commodity SEC-DED codes.

Outline. The paper is structured as follows: Section 2 discusses the required background. Section 3 elaborates on combining integrity protection, authenticated encryption, and memory tagging. Section 4 presents our novel tag encodings and their properties. Section 5 discusses and simulates the error detection and correction capabilities of `Voodoo`. Section 6 details our implementation and showcases the advantages of our combined mode through benchmarking. Section 7 discusses related work, and Section 8 summarizes our findings.

2 Background

This section provides background on authenticated encryption, DRAM structure and faults, memory tagging, and `MAGIC`.

2.1 Authenticated Encryption

Encrypting data grants *confidentiality* for the data, as only a party possessing the correct key can encrypt and subsequently read the data. However, an adversary can still modify encrypted data. Altering a single bit of a ciphertext will cause the resulting plaintext to deviate from the originally encrypted plaintext. Even though such an attack cannot change the plaintext precisely, it can still entail security issues [37, 61].

With *authenticated encryption* (AE), unprivileged ciphertext modifications are detected. Authenticated encryption provides *integrity* and *authenticity* through a message authentication code (MAC). A MAC is a mapping $\mathcal{M} : F_2^* \times F_2^k \rightarrow F_2^t$. An arbitrarily sized input and a secret k-bit key map to a t-bit checksum as the output. Only a party possessing the key can compute a correct MAC for a given input. When accessing data, recomputing the MAC and comparing it with a stored version allows for the detection of data modifications.

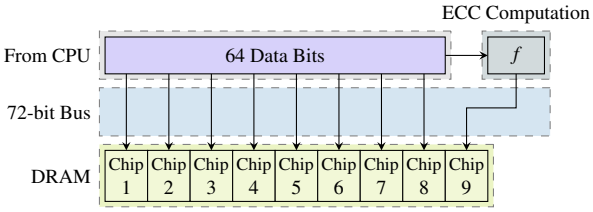

Figure 1: Writing data using an ECC DRAM. The function f computes ECC bits that are stored alongside the data on a separate chip on the DRAM module.

2.2 DRAM Structure and Faults

DRAM modules store data in *memory cells*. Each cell holds one bit in the form of a charged or discharged capacitor. Memory cells are organized in *rows* and *columns*, forming a *memory array*. Multiple (sub-)arrays are combined to *banks*, which are further organized in *bank groups*. Banks operate in parallel, thus improving the performance of the memory. A single DRAM chip can host multiple banks. We distinguish between x4, x8, and x16 DRAM chips [25]. The number defines the amount of data pins of the chip. Multiple DRAM chips form a *rank*, and, in the case of DDR4 DRAM, the overall data bus width is 64 bits. DRAM accesses are performed in *bursts* where each burst yields 512 bits of data. For DDR4 DRAM with a 64-bit bus width, each burst consists of 8 subsequent accesses called *beats*.

Memory cells are inherently error-prone as the stored charges are subject to leakage [24, 46]. Thus, they must be refreshed periodically [25]. Excessive leakage causes the charge to be unrecoverably lost. Hence, the stored data can become faulty. Besides leakage, external disturbances can influence the charge stored in DRAM cells [4, 38]. While single-bit faults are the most common fault, multiple bits read from a single DRAM chip may be erroneous [5, 36, 49, 53].

To mitigate the effects of DRAM faults, error-correcting codes (ECC) are used [14, 24]. An ECC extends the data with redundancy to restore the data in the case of a fault. For DDR4 memory with ECC, each 64-bit memory word is accompanied by 8 ECC bits, resulting in a storage overhead of 12.5%. The redundant data is stored in additional chips on the DRAM module. ECC DRAM modules have a 72-bit wide bus to allow simultaneous fetching of data and redundancy. Figure 1 illustrates how the data and the redundant information is stored to a DRAM module using nine x8 chips. One class of commodity ECC are single-error correction, double-error detection (SEC-DED) codes [19]. With SEC-DED, correcting one and detecting two errors per 64-bit word is possible. Faults affecting more than two bits in a 64-bit word are not guaranteed to be detected. Undetected faults may be miscorrected and cause the system to process faulted data. Chipkill codes offer stronger protection, as they can correct up to a complete chip failure [14].

2.3 Memory Tagging

Tagged memory [26] enforces different security policies for memory safety [50, 51, 58], memory isolation [48, 52, 59, 62, 64], capability systems [8, 63], or dynamic information-flow tracking (DIFT) [13, 28, 54]. Every memory location is associated with metadata, the so-called *memory tag*, which has a specific *tag size* and *tag granularity*. Capability architectures, such as the M-Machine [8] and CHERI [63], use a single-bit tag to mark and protect capabilities in memory. CHERI associates a single-bit tag with every 16 or 32 bytes of memory.

Commercial products like the ARM memory tagging extension (MTE) [50] associate each 16-byte granule with a 4-bit tag [51], allowing fine-grain policies. A pseudorandomly generated tag is encoded into the upper bits of each pointer. The dedicated tag is associated with the corresponding memory location through the tagged memory architecture. Access to the memory location is only granted if the pointer tag matches the tag of the location. Similarly, SPARC application data integrity (ADI) [1] uses a 4-bit memory tag at the granularity of 64 bytes for memory access checks [51].

Memory tagging can also be used for isolation. Mondrian [62] uses 2-bit tags per word, while Loki [64] uses up to 32 bits per word to enforce memory isolation. HDFI [52] enforces data-flow isolation with a single-bit memory tag. Furthermore, memory tagging can be used to design enclave architectures. SPEAR-V [48], for example, uses memory tagging on page granularity to isolate enclave memory. TIMBER-V [59] allows for stack interleaving based on fine-grain memory tagging using 2-bit tags per word.

While tagged memory architectures are versatile and help to improve runtime security, they share common drawbacks if implemented naïvely. Tags require additional storage space that scales with the tag size and granularity. Thus, the size of usable system memory declines. Furthermore, the tags must be fetched and propagated by the tagged memory architecture. This causes additional pressure on the DRAM bus, resulting in a loss of performance [27]. Especially for workloads typically encountered in servers and cloud systems, additional DRAM traffic has a strong performance impact [27, 52].

2.4 The `MAGIC` Mode

In 2020, Kounavis et al. proposed the `MAGIC` mode for authenticated encryption and error correction, bridging the gap between authenticated encryption and DRAM fault protection [31]. `MAGIC` computes an N-bit checksum T over n N-bit ciphertext blocks C_i and an N-bit block of authenticated data D. During the computation, `MAGIC` uses a blinding cipher E_K, where K denotes the key used by the encryption. Similar to `GMAC`, `MAGIC` uses a secret N-bit *hash key* H [43]. The checksum is computed as

$$T = E_K\left(D + \sum_{i=1}^{n} C_i \cdot H^i\right) \quad (1)$$

using finite field arithmetic in GF(2^N). When reading from memory, MAGIC computes an N-bit syndrome S as

$$S = E_K^{-1}(T) + D + \sum_{i=1}^{n} C'_i \cdot H^i$$

where C'_i denotes the ciphertext block read back from DRAM. If one of the ciphertext blocks at index j was modified, then $C_j \neq C'_j$ and the syndrome will evaluate to $S = (C_j + C'_j) \cdot H^j = e_j \cdot H^j$. MAGIC corrects errors with a Hamming weight less than or equal to a chosen threshold T_{th} and confined to a single ciphertext block. T_{th} determines the maximum number of correctable bits. Checksum differences with a Hamming weight below T_{th} are considered to be an error on the checksum and are corrected by replacing the faulted checksum. Values for H are selected such that

$$(\forall e : e \neq 0, \mathrm{HW}(e) \leq T_{th}) \wedge (\forall i \in [1, n-1]) : \\ \mathrm{HW}\left(H^i \cdot e\right) > T_{th}, \mathrm{HW}\left(H^{-i} \cdot e\right) > T_{th} \quad (2)$$

holds. When performing error correction, MAGIC locates the erroneous ciphertext block by computing the *error location indicators* S_i as $S_i = S \cdot H^{-i}$. Equation (2) guarantees that $\mathrm{HW}(S_i) \leq T_{th}$ holds only if i equals the index of the faulted ciphertext block. The error can be corrected by XORing the error location indicator to the faulted block.

The error correction of MAGIC stems from carefully selecting H, which is not randomly drawn but chosen according to Equation (2). This condition limits the number of permissible values of H. Let $\mathcal{H}$ be the set of values H that fulfill Equation (2). The cardinality of $\mathcal{H}$ can be estimated as

$$|\mathcal{H}| \geq 2^N - \frac{n(n-1)}{2} \left(\sum_{t=1}^{T_{th}} \binom{N}{t} \right)^2 . \quad (3)$$

Equation (3) helps to estimate whether some valid H values are guaranteed to exist. A positive result guarantees the existence of permissible values for H. In general, one can find permissible values by sampling a value at random and asserting that Equation (2) holds. With $N = 64$ bits, $n = 8$ bits and $T_{th} = 7$ bits, for example, the set $\mathcal{H}$ contains at least $2^{61.9}$ values.

3 Combining Integrity Protection, Authenticated Encryption, and Memory Tagging

In this section, we detail how Voodoo extends on MAGIC to implement tagged memory through the output produced by the MAC computation. First, we identify the requirements for a combined authenticated encryption, error correction, and memory tagging scheme. Then, we discuss the principle design of Voodoo. We elaborate on the functionality and the involved hardware components that process the data and the memory tag. Finally, we discuss the drawbacks and pitfalls of a naïve memory tagging implementation without sufficient tag encoding.

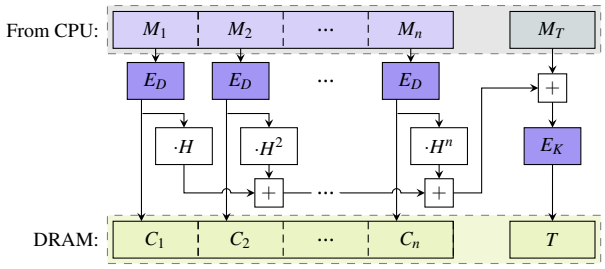

Figure 2: Overview of the Voodoo hardware architecture. The data $M_1, \cdots, M_n$ is first encrypted and then used as the input for the MAC computation. The encoded memory tag M_T is added to $\sum_{i=1}^{n}(C_i \cdot H^i)$, thus forming the intermediate of the MAC computation.

For the sake of clarity we will refer to the output of the MAC computation as the *checksum*. The metadata used for memory tagging is referred to as the *memory tag* or *tag*. When including a memory tag M_T into the checksum T we must consider how the memory tag may interact with the other features of our scheme. We identify the following three requirements that we aim to fulfill.

Accesses with a wrong tag must be detected. Memory tagging aims to improve the runtime security of a system. A fundamental property of tagged memory architectures is that memory accesses with an incorrect tag are detected. Thus, Voodoo must reliably detect tag mismatches. As tagging implements security-critical policies, a tag mismatch will terminate the process that encounters the error.

Error detection and correction are not weakened significantly. Using the checksum T to hold metadata for memory tagging influences error detection and correction. Thus, the encoding must not significantly weaken the error detection and correction abilities.

The memory tag can be read. For certain use cases, such as swapping and process migration, it is necessary to read the tag associated with a memory location. As the memory allocator usually performs tagging, we cannot assume that the operating system has knowledge of which memory is associated with which tag. Thus, we aim to support extracting memory tags from checksums.

3.1 Design Overview

Voodoo consists of two key components. The first component is the hardware architecture depicted in Figure 2. The second component is the *memory tag encoding* discussed in Section 4. The encoding does not necessarily require a hardware building block. Unless stated otherwise, M_T is already encoded when written to memory.

Checksum and Syndrome Computation. In general, we compute the checksum when writing to memory as

$$T = E_K\left(M_T + \sum_{i=1}^{n} C_i \cdot H^i\right) . \quad (4)$$

We denote the value passed to the blinding cipher E_K as the *intermediate*. When reading from memory, we compute the syndrome as

$$S = E_K^{-1}(T) + M'_T + \sum_{i=1}^{n} C'_i \cdot H^i. \tag{5}$$

Here, M'_T denotes the memory tag provided during the read operation, and C'_i represents the ciphertext blocks as read from memory. All operations are performed in $GF(2^N)$.

The checksum computation is depicted in Figure 2. First, the data written to memory is split into n N-bit blocks and encrypted using a data encryption cipher E_D. E_D can be any suitable cipher whose output matches the block size of N bits. The encrypted data C_1 to C_n and the memory tag M_T are then used to generate the intermediate value of the MAC computation. Applying the blinding cipher E_K to the intermediate yields the checksum T. We assume that an ECC DRAM is used. Hence, the ciphertext and the checksum can be stored and fetched within one access. Using a tweakable block ciper such as QARMA [3] for E_D allows to include the physical address as an encryption tweak. Thus, the same data at different locations in memory yields different ciphertexts and different checksums. AMD SEV-SNP, for example, uses this technique to protect againt attacks moving ciphertext in memory [61].

The values of H are secret and must fulfill Equation (2). However, we tighten the condition by requiring the products $e \cdot H^i$ with exponents *up to n* to produce a Hamming weight strictly above T_{th}. The condition is modified to

$$\begin{aligned} &(\forall e : e \neq 0, \mathrm{HW}(e) \leq T_{th}) \wedge (\forall i \in [1,n]) : \\ &\mathrm{HW}\left(H^i \cdot e\right) > T_{th}, \mathrm{HW}\left(H^{-i} \cdot e\right) > T_{th} \end{aligned} \tag{6}$$

The condition used by MAGIC (Equation (2)) does not ensure $\mathrm{HW}(S) > T_{th}$, as i never reaches n. With Equation (6), every syndrome from a correctable error has a Hamming weight above T_{th}.

Error Detection and Correction. When reading from memory, the syndrome is computed as given in Equation (5). A non-zero syndrome indicates a data error or a tag mismatch. In the case of a data error, the error correction procedure is the same as for MAGIC. We compute the error location indicators as $\forall i \in [1,n] : S_i = S \cdot H^{-i}$. An error is deemed correctable *iff* precisely one of the error location indicators has a Hamming weight that is not above T_{th}. Let j denote the index of the erroneous block identified through the error location indicators. The error is fixed by computing $C_j = C'_j + S \cdot H^{-j}$. If more than one error location identifier shows a Hamming weight that is in range, we consider this to be an uncorrectable error. If no index j is found, we also treat the error as an uncorrectable error.

Errors that affect the checksum T can be corrected by computing the Hamming weight of the difference between the fetched and the computed checksum. If the Hamming weight is less than a pre-defined threshold, we assume that the checksum suffered a fault and restore the correct checksum by overwriting it with the computed checksum.

Table 1: The misinterpretation probability of a naïve tag encoding w.r.t. the number of correctable bits. N and T_{th} are given in bits.

N \ T_{th}	1	2	3	4	5	6	7
64	2^{-55}	2^{-49}	2^{-45}	2^{-41}	2^{-38}	2^{-34}	2^{-31}
128	2^{-119}	2^{-112}	2^{-107}	2^{-102}	2^{-97}	2^{-93}	2^{-89}

3.2 Naïve Tag Encoding

In general, MAGIC supports the authentication of additional data when computing T. However, this additional data without any encoding is not suitable for implementing memory tagging. Assume that M_T can be freely selected from $\{0,1\}^N$. We denote this encoding as f-Unbounded. The modified checksum is computed as described in Equation (4). Including the memory tag in the checksum guarantees that an access with a wrong memory tag causes a non-zero syndrome. Assume the case in which we perform an access using a wrong memory tag M'_T. The syndrome will evaluate to $S = M_T + M'_T = e_T$. This value is equal to the *difference between the two memory tags* and will never be zero. Thus, the mismatch is detected. However, the syndrome cannot be distinguished from a syndrome that is due to errors in the ciphertext blocks.

A tag mismatch is identified correctly if $S = e_T$ is not interpreted as a correctable error. An e_T that causes exactly one error location indicator to have a Hamming weight less than or equal to T_{th} will be treated as a correctable error. We denote the case in which a memory tag mismatch causes error correction as a *misinterpretation*. Let $\mathcal{C}_f$ be the set of error location indicators that will be misinterpreted as a correctable error due to their low Hamming weight. Given an N-bit block size, the cardinality of $\mathcal{C}_f$ is given as

$$|\mathcal{C}_f| = \sum_{t=1}^{T_{th}} \binom{N}{t}. \tag{7}$$

For a misinterpretation, exactly one error location identifier must have a Hamming weight less than or equal to T_{th}. Using Equation (7), we can compute the misinterpretation probability. Given a memory tag error, the probability P_{Mis} that the mismatch is interpreted as a correctable error is

$$P_{Mis} = n \cdot \left(\frac{|\mathcal{C}_f|}{2^N}\right) \cdot \left(1 - \frac{|\mathcal{C}_f|}{2^N}\right)^{n-1}. \tag{8}$$

Memory tag errors with a Hamming weight lower than or equal to T_{th} will never be falsely interpreted due to Equation (6). Table 1 lists the misinterpretation probabilities for

Table 2: The aliasing probability of f-Pattern for X tag bits for $N = 64$ bits and $N = 128$ bits, respectively. T_{th} is the Hamming weight threshold in bits.

N	T_{th} \ X	1	2	4	8	16	32
64	7	0	0	0	2^{-63}	2^{-48}	2^{-32}
128	14	0	0	0	0	2^{-123}	2^{-96}

different T_{th}. As the cardinality of $\mathcal{C}_f$ increases, so does the misinterpretation probability. Due to the chance of misinterpretation, this encoding cannot be used for deterministic memory tagging. Even worse, some tag mismatches can cause silent data corruption.

Memory Tag Extraction. In the case that no error occurs in the ciphertext blocks, we can extract the memory tag from the checksum. Let M'_T denote the memory tag that we extract from T. We can compute $M'_T = E_K^{-1}(T) + \sum_{i=1}^{n} C'_i \cdot H^i = M_T$. It is, however, impossible to extract the correct tag in the case that an error occurred. Assume that the ciphertext block with index j is faulty and let e_j denote the error of the block. Performing the above computation will yield $M'_T = M_T + e_j \cdot H^j$. Thus, the extracted memory tag will not equal the previously encoded tag but contain the error as well. As the possible range for memory tags is not constrained, it is impossible to detect this error.

4 Encoding Tags for `Voodoo`

Based on the requirements and architecture given in Section 3, we develop novel memory tag encodings that meet the requirements while avoiding the issues and pitfalls of f-Unbounded. We introduce three novel encoding schemes called *Check Pattern Encoding* (f-Pattern), *Encrypted Tag Encoding* (f-Encrypt), and *Bounded Hamming Weight Encoding* (f-Bounded). For each encoding, we analyze the impact on the error detection and correction of `Voodoo`. Furthermore, we discuss how memory tags can be extracted from the checksum, even in the presence of errors. In the following, we denote the set of possible memory tags as $\mathcal{M}_T$. We denote memory tag differences as e_T and the encoded memory tag as M_T. For tag extraction, we denote the extracted memory tag as M'_T.

4.1 Check Pattern Encoding

In the naïve encoding, tags can be drawn from the full N-bit range. Most tagged memory architectures do not require such a wide range of tags. ARM MTE, for example, uses 16 tag bits for 512 data bits distributed over four 128-bit granules [51]. As our underlying encoding approach allows for N tag bits, 48 bits would remain unused in the case of MTE for $N = 64$ bits. We can use these spare bits to encode a fixed pattern into the memory tag. When accessing data with a wrong tag, the fixed pattern cancels out when computing e_T. Thus, the syndrome S will only be non-zero in the bits that are actually used by the tagged memory architecture.

The pattern does not directly decrease the odds of a misinterpretation. However, we can modify the correction procedure such that a misinterpretation of tag errors is impossible. In the original correction procedure, we search for the error location using the error location indicators. We modify the procedure by initially checking whether the upper bits of the syndrome are zero. If all upper bits (*i.e.*, the unused bits of the tag encoding) are zero, we assume the error to be a memory tag error. Such errors are treated as uncorrectable errors. The modified procedure will never misinterpret a tag mismatch as the error correction is aborted. Thus, a spurious correction of data that did not experience a fault is impossible. However, we experience an increased rate of uncorrectable errors.

Assume an encoding where only X bits out of N bits are used for the actual tag. The remaining bits are set to a fixed pattern. With an X-bit tag, we can express at most 2^X tag differences with a maximum Hamming weight of X. Thus, a syndrome that fulfills $S \geq 2^X$ is never caused by a tag mismatch. Let $\mathcal{U}$ denote the subset of all possible tags with a Hamming weight that is less than or equal to T_{th}. The elements of $\mathcal{U}$ are unproblematic as no correctable error will ever map to this subset due to Equation (6). The cardinality of $\mathcal{U}$ is $|\mathcal{U}| = \sum_{t=0}^{T_{th}} \binom{X}{t}$.

The syndrome produced by a correctable error can be one of $2^N - \sum_{t=0}^{T_{th}} \binom{N}{t}$ possible values, as the syndrome will always have a Hamming weight above T_{th}. Then, the probability P_{alias} that a correctable error maps to one of the remaining (problematic) values can then be given as

$$P_{alias} = \frac{2^X - |\mathcal{U}|}{2^N - \sum_{t=0}^{T_{th}} \binom{N}{t}}. \tag{9}$$

Note that if X is less than T_{th} bits, the aliasing probability becomes 0, as no error will ever map to a valid tag difference. Table 2 lists the aliasing probabilities for $N = 64$ bits and $N = 128$ bits depending on the desired tag size X. The values chosen for T_{th} are the largest possible values for their respective N such that valid H values are guaranteed to exist (c.f. Equation (3)). This encoding allows for deterministic memory tagging *without aliasing* if the chosen tag size is small enough to be covered by the correction threshold. For large tag sizes, however, the encoding may become unsuitable as the probability of uncorrectable errors increases.

Memory Tag Extraction. With f-Pattern, it is possible to extract the memory tag even in the case of a data error that is present in one of the ciphertext blocks. Due to the distinct pattern of valid tags, we can most likely identify faults during the tag-read operation. The probability that a data error has the shape of a valid tag is equal to the aliasing probability defined in Equation (9). If $X \leq T_{th}$, this probability equals 0, indicating that a data error cannot be confused with

a memory tag. If a data error aliases to a valid tag, the tag extraction will produce a faulty memory tag. When a data error is correctly identified as such, we can try to restore the tag and correct the fault. Computing $M'_T = E_K^{-1}(T) + \sum_{i=1}^{n} C'_i \cdot H^i$ yields $M'_T = e_j \cdot H^j + M_T$. The error e_j can be corrected by applying all 2^X possible tag values to this intermediate result and performing the regular error correction procedure. For the correct memory tag, it is guaranteed that the correction procedure will succeed. However, multiple memory tag values may lead to correctable errors. To avoid miscorrection, we always test all possible memory tags and only correct the fault if precisely one of the memory tags leads to a successful error correction. Once the data error is corrected, the memory tag can be extracted by reading out the syndrome.

4.2 Encrypted Tag Encoding

A different approach to improve the shortcomings of the naïve encoding is by applying an additional encryption on the memory tag M_T before encoding it into the checksum T. With f-Encrypt, we encrypt the memory tag using a suitable block cipher, and Equation (4) is transformed to

$$T = E_K\left(E_T(M_T) + \sum_{i=1}^{n} C_i \cdot H^i\right). \tag{10}$$

We eliminate the case of data corruption due to tag mismatches by imposing an additional constraint on how the values of M_T are conditioned. We limit the memory tags only to values that have either low or high Hamming weights. Formally, we specify that $\mathrm{HW}(M_T) \leq L_L \vee \mathrm{HW}(M_T) \geq L_U$ must hold where L_L and L_U denote the lower and upper Hamming weight limits, respectively. The size of the available tag space can be computed as $|\mathcal{M}_T| = \sum_{t=0}^{L_L} \binom{N}{t} + \sum_{t=L_U}^{N} \binom{N}{t}$. Encrypting a memory tag from this tag space will lead to a ciphertext uniformly drawn from $\{0,1\}^N$. In the case of a tag mismatch, the resulting difference will again be uniformly drawn. The probability that such a difference is misinterpreted as a correctable error is the same as for f-Unbounded. However, we can use the fact that actual memory tags have either low or high Hamming weights to distinguish between memory tag mismatches and actual data errors. Assume a tag difference e_T causes an error location indicator to have a Hamming weight less than or equal to T_{th}. We can now compute $M'_T = E_T^{-1}\left(E_K^{-1}(T) + \sum_{i=1}^{n} C'_i \cdot H^i\right)$ in an attempt to restore the memory tag. In the case of an actual data error, the computation will evaluate to $M'_T = E_T^{-1}\left(E_T(M_T) + e_j \cdot H^j\right)$. In the most probable case, the result of this computation has a Hamming weight above L_L and below L_U. The probability P_{HW} that a non-zero syndrome originating from an error in the data maps to a valid tag when decrypted can be given as

$$P_{\mathrm{HW}} = \frac{\left(\sum_{t=0}^{L_L} \binom{N}{t} + \sum_{t=L_U}^{N} \binom{N}{t}\right)}{2^N}. \tag{11}$$

The unlikely case that the result suggests an error due to a tag mismatch is treated as an uncorrectable error. With this modification, it is impossible for a tag mismatch to lead to miscorrection, and every tag mismatch is guaranteed to be detected. As a tradeoff, we increase the probability that a correctable error is interpreted as a tag mismatch and treated as an uncorrectable error. Equation (11) allows us to tune this probability by adjusting L_U and L_L. While the encryption of the memory tag adds additional latency, the actual performance impact is low. Memory tags are encrypted when writing to memory, which can happen parallel to the encryption of the plaintext data. Thus, encrypting the memory tag does not add latencies to the MAC computation. Decrypting the encrypted memory tag is only necessary if the syndrome does not evaluate to zero. No additional latency is introduced for read accesses without tag mismatches or errors, which is the common case.

Memory Tag Extraction. When using f-Encrypt, we compute $M'_T = E_T^{-1}\left(E_K^{-1}(T) + \sum_{i=1}^{n} C'_i \cdot H^i\right)$. In the error-free case, this will always yield the correct tag value. Given that a data error occurred, the result of the outermost decryption will be a uniformly drawn, random value. The probability that this decryption leads to a valid tag is equivalent to the probability given in Equation (11). As with f-Pattern, we can try to correct the error and then restore the memory tag by iteratively searching for the correct memory tag and performing the error correction procedure. If precisely one memory tag leads to a correctable error, we have found the erroneous ciphertext block and can extract the tag.

4.3 Bounded Hamming Weight Encoding

We can completely avoid misinterpretation and aliasing by limiting the values of M_T such that the condition $\mathrm{HW}(e_T) \leq T_{th}$ is fulfilled for all possible memory tag differences e_T. Such memory tags can never alias to correctable errors due to Equation (6). Contrary to f-Pattern and f-Encrypt, we do not need to modify the error correction procedure to distinguish between tag mismatches and data errors. Instead, searching for the error location will yield an inconclusive result as the Hamming weight of e_T is always less than or equal to T_{th}. Legitimate errors in data blocks will always lead to Hamming weights above T_{th}, as the selection of H ensures this property. Thus, by encoding the memory tags such that $\mathrm{HW}(e_T) \leq T_{th}$ is fulfilled for all possible tags, we can deterministically distinguish between memory tag mismatches and actual errors in the ciphertext blocks. However, the encoding limits the possible set of memory tags to guarantee that the difference between two memory tags is always within the threshold.

Assuming that each block in the MAC computation is N bits wide, we can compute the number of possible tag values given the value of T_{th}. Each memory tag may have a maximum Hamming distance of T_{th} to any other memory tag; hence the maximum number of set bits per memory tag equals $\lfloor T_{th}/2 \rfloor$.

Table 3: The available tag space in bits (Equation (12)) for $N = 64$ bits and $N = 128$ bits, respectively, when using f-Bounded. T_{th} is the Hamming weight threshold in bits.

N \ T_{th}	2	4	6	8	10	12	14
64	6	11	**15.4**	-	-	-	-
128	7	13	18.4	23.3	28	32.4	**36.5**

It follows that

$$|\mathcal{M}_\mathcal{T}| = \sum_{t=0}^{\lfloor T_{th}/2 \rfloor} \binom{N}{t}. \tag{12}$$

Table 3 shows how the available tag space increases with an increasing T_{th}. While bounding the Hamming weight limits the tag space, it allows us to encode tags without the chance of misinterpretation or aliasing. Thus, we deterministically detect every tag mismatch without increasing the probability of uncorrectable errors.

Memory Tag Extraction. For f-Bounded, it is impossible that a data error assumes the shape of a tag error. Memory tags have a maximum Hamming weight of $\lfloor T_{th}/2 \rfloor$. According to Equation (6), errors in the ciphertext blocks will always lead to a product with a Hamming weight strictly above T_{th}. When computing $M'_T = M_T + e_j \cdot H^j$ we can always detect the presence of an error. When adding two elements $a, b \in GF(2^N)$ with $\mathrm{HW}(a) > \mathrm{HW}(b)$ the Hamming weight of the result is at least $\mathrm{HW}(a) - \mathrm{HW}(b)$ The resulting Hamming weight of M'_T is, thus, guaranteed to be at least $\lfloor T_{th}/2 \rfloor + 1$. Hence, the result can never have the shape of a valid tag difference. Like f-Pattern, we can recover the tag and correct the error by testing all possible tag values and performing the error correction procedure. If precisely one tag leads to a correctable error, we have successfully restored the tag and corrected the data. If multiple tags would lead to an error correction, we consider the data to be uncorrectable.

Comparison of Encodings. Table 4 summarizes the capabilities of the presented encoding schemes. f-Unbounded is not suitable to implement tagged memory. Memory tag mismatches are not guaranteed to be detected and can cause miscorrection, thus violating data integrity. Furthermore, it is impossible to read the memory tag in the presence of a data error. All other encoding schemes guarantee that tag errors are detected and never cause miscorrection. They each provide a mechanism to read tags, even in the presence of data errors. However, f-Pattern and f-Encrypt may return a faulty tag in the case that a data error is present during tag extraction. With f-Bounded, we reach the strongest capabilities, as this scheme offers determinism in all analyzed aspects. It is impossible that f-Bounded returns a faulty memory tag during tag extraction.

Table 4: A summary of the capabilities of the different encoding schemes and their success probabilities.

Encoding	Detect Mismatch	Correct Error	Read Tag w. Error
f-Unbounded	$P = 1 - P_{mis}$	✓	✗ read wrong tag
f-Pattern	✓	$P = 1 - P_{alias}$	$P = 1 - P_{alias}$
f-Encrypt	✓	$P = 1 - P_{\mathrm{HW}}$	$P = 1 - P_{\mathrm{HW}}$
f-Bounded	✓	✓	✓

$$isTagMismatch(S) = \begin{cases} HW(S) \leq T_{th}, & \text{if } f\text{-Bounded.} \\ S < 2^X, & \text{if } f\text{-Pattern.} \\ HW(E_T^{-1}(S + E_T(M'_T))) \notin]L_L, L_U[, & \text{if } f\text{-Encrypt.} \\ \bot, & \text{if } f\text{-Unbounded.} \end{cases} \tag{13}$$

Figure 3 shows the generic flow of a read operation with Voodoo. Depending on the encoding pattern, Equation (13) decides if a S value looks like a tag, which indicates that a tag mismatch happened, in which case we do not correct the error.

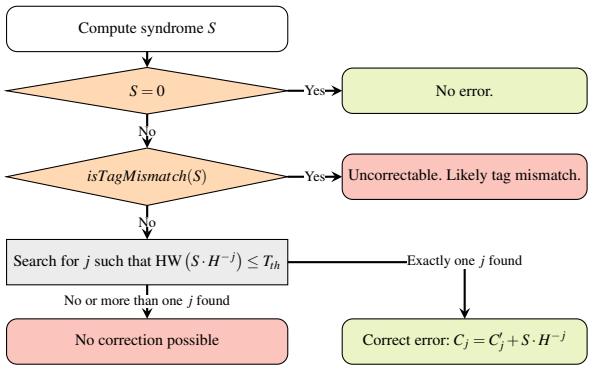

Figure 3: Flow of a DRAM read operation with Voodoo.

4.4 Case Study on Tagged Architectures

In this section, we present a case study that demonstrates Voodoo's applicability. We investigate which tagged memory architectures can be implemented with our encodings. Our findings indicate that Voodoo is suitable for a wide range of academic and commercial tagged architectures. We consider a tag encoding suitable if the available tag space can accommodate the required tag size of the memory tagging scheme. For f-Encrypt, we assume a configuration in which T_{th} is set to the maximum value for the respective block size N. We select L_U and L_L such that the available tag space is minimized but large enough to cover the required tag space. For f-Pattern, we assume that X is chosen to equal the required tag size of the memory tagging scheme. In addition to existing architec-

tures, we analyze models with theoretical tag sizes to show the limitations of our approach.

Supported Tag Sizes. Several architectures, including CHERI [63], DIFT [26], and HDFI [52], implement a single-bit tag per granule. f-Bounded presents an ideal solution for schemes requiring a small tag space. CHERI's 256-bit and 128-bit version yield 2-bit and 4-bit tags per 64 B cache line, respectively. Both variants can be deterministically implemented using f-Bounded. If the parameters are set accordingly, f-Pattern also offers deterministic tagging without aliasing for both granularities. With f-Encrypt, there is always the possibility that data errors alias to tag mismatches. Thus, we can implement both variants but must accept a slight increase in the rate of uncorrectable errors.

Other single-bit memory tagging schemes, such as DIFT or HDFI, operate at a word level of granularity. With 64-bit words, this results in 8-bit memory tags per 64 B cache line. While this tag space is still deterministically covered by f-Bounded, configurations of f-Pattern with $N = 64$ already experience aliasing with a small probability. With f-Encrypt and $N = 64$, we cannot select the threshold values to reach a tag space of exactly 8 bits. Instead, we must use the next larger size of 11 bits. This comes with the downside of an increased rate of uncorrectable errors. For $N = 128$ bits we can select L_U and L_L to reach a tag space of 8.01 bits.

Memory safety countermeasures like ARM MTE [50] and SPARC ADI [1] rely on multi-bit memory tags. MTE and ADI utilize a 4-bit memory tag with a respective granularity of 16 B and 64 B. Thus, SPARC ADI requires four tag bits per cache line, and ARM MTE needs a larger 16-bit tag space. While f-Bounded can cover the tag space of SPARC ADI, it is not possible to provide all tags needed by ARM MTE in the case of $N = 64$ bits. With $N = 128$ bits, both architectures can be implemented deterministically. Also, both architectures can be implemented using f-Pattern and f-Encrypt while offering a reasonably low probability of aliasing.

Some memory isolation schemes [12, 48] require larger memory tags. SPEAR-V [48], for example, uses 24-bit memory tags on page granularity. As our design operates on cache line granularity, we need to encode the memory tag in every cache line, resulting in a tag size of 24 bits. The f-Bounded mode is not applicable here for $N = 64$ bits. However, with $N = 128$ bits we can deterministically encode the tag. With f-Pattern and f-Encrypt, we can encode the complete 24-bit tag but suffer an increased probability of uncorrectable errors.

Both f-Pattern and f-Encrypt offer a tradeoff between deterministic detection of tag mismatches and error correction capabilities. Depending on the expected rate at which data errors occur, a slight increase in uncorrectable errors might be acceptable. However, the encodings may not be suitable for architectures with large tag sizes, as the rate will be too high, and the system will become unreliable.

Supported Architectures. Depending on the tagged memory architecture, memory accesses either carry a tag value that is used for comparison, or the tag value must be extracted from the syndrome. `Voodoo` supports both types of tagged memory architecture, as our encodings allow for tag extraction, even in the presence of errors. However, architectures that do not present a tag on each access (e.g., , DIFT) may suffer from additional latencies when accessing memory locations that suffered a data error. The additional latency is determined by the number of possible memory tags, as they are restored using an iterative approach. For the latency computation, we multiply the size of the tag space with the latency of a single trial computation, which we approximate with 0.44 ns [3]. We find that most tagged architectures are feasible to implement with `Voodoo`. For architectures that use large tags, such as SPEAR-V, the latency can become rather high. As this latency is *only present in the first access to an erroneous data block*, the overall impact is still low. For architectures such as lowRISC or Model 2B, the reconstruction latency is infeasibly high. Thus, architectures with large tags are expected to perform better if the tag is present on each memory access.

Table 5: Case study of our tag encodings. The numbers indicate the probability that a tag mismatch maps to an uncorrectable error. ● indicates deterministic tagging and ○ indicates an impossible configuration. Architectures that present a tag on every access are marked with 🏷, architectures that require tag extraction are marked with 🔍. All block sizes are given in bits. For architectures that do not present a tag on every access, we give the additional latency that occurs when reading the tag from an erroneous location.

Architecture	Tag Bits per 64 B	f-Bounded N=64	f-Bounded N=128	f-Pattern N=64	f-Pattern N=128	f-Encrypt N=64	f-Encrypt N=128	Latency in ns
CHERI ISA (256) [63]🏷	2-bit	●	●	0	0	2^{-57}	2^{-120}	-
CHERI ISA (128) [63]🏷	4-bit	●	●	0	0	2^{-57}	2^{-120}	-
SPARC ADI [1]🏷	4-bit	●	●	0	0	2^{-57}	2^{-120}	-
DIFT [26]🔍,M-Machine [8]🏷, HDFI [52]🏷,Shakti-T [44]🏷	8-bit	●	●	2^{-63}	0	2^{-52}	2^{-119}	$0.1 \cdot 10^3$
Model 1 A🏷 / B🔍	15-bit	●	●	2^{-49}	2^{-128}	2^{-48}	2^{-109}	$14 \cdot 10^3$
MTE [50]🏷,Mondrian [62]🔍	16-bit	○	●	2^{-48}	2^{-123}	2^{-47}	2^{-109}	$29 \cdot 10^3$
SPEAR-V [48]🔍	24-bit	○	●	2^{-40}	2^{-106}	2^{-37}	2^{-103}	$7 \cdot 10^6$
lowRISC [39]🔍,🏷	32-bit	○	●	2^{-32}	2^{-96}	2^{-31}	2^{-95}	$2 \cdot 10^9$
Model 2 A🏷 / B🔍	36-bit	○	●	2^{-28}	2^{-92}	2^{-26}	2^{-91}	$30 \cdot 10^9$

Table 5 provides a concise case study overview. Tagged memory architectures are sorted according to their tag sizes. All exponents given for the aliasing probabilities are rounded towards higher probabilities. For f-Bounded, a full circle indicates that we can offer deterministic tagging, while an empty circle indicates that the encoding is unsuitable. For f-Pattern and f-Encrypt, we list the corresponding aliasing probabilities for the optimal parameterization described above. As f-Encrypt does not allow us to specify the tag size directly, we must select the parameters such that the available tag space covers the required tag space.

5 DRAM Error Detection and Correction

This section discusses the DRAM error detection and correction of Voodoo. We introduce a fault model based on DRAM faults in DDR4 devices. We then discuss the error detection and correction capabilities for single-block and multi-block errors. By bounding chip errors to single ciphertext blocks, we show how to improve the error correction of Voodoo. Furthermore, we implement a Monte-Carlo simulation to showcase the error detection and correction capabilities. Lastly, we compare the error correction performance to that of commodity and academic error correction schemes.

5.1 Fault Model

We base our fault model on real-world data produced by a large-scale DDR4 fault study of Beigi et al. [5]. Their work identifies sixteen fault modes that affect DDR4 devices. Table 6 lists a coarse-grain overview of the modes we consider in our analysis. The combined fault modes are derived from the fault modes identified by Beigi et al. [5]. We assume that a burst consists of 8 beats where each beat transfers 64 data bits over the bus [25]. We assume that each rank on the DIMM hosts 16 x4 DRAM devices (cf. Section 2) storing data and two x4 DRAM devices for storing redundancy, as this is the setup used in the study of Beigi et al. [5]. Thus, each x4 DRAM device will contribute exactly 32 data bits to the 512 data bits that comprise a cache line. We consolidate modes that will affect the same worst-case number of bits together for simplicity. We identify four fault classes ranked according to the maximum number of affected bits.

Single-Bit Faults. Single-bit faults are the most common faults observed in modern DRAM devices. Here, the content of a single memory cell is corrupted, and only a single bit of the data read from the DRAM device is erroneous.

Multi-Bit Faults. Multi-bit faults affect two to four bits in a single burst. This fault class combines single-word faults and single-column faults.

Subsequent Faults. The class of subsequent faults captures all faults where two of the beats in a burst yield faulted data. The maximum number of faulty bits in this fault mode is 8, as each beat yields 4 bits of data in the case of DRAM chips with four data pins. This fault class consists of two-column faults and single-pin faults.

Large-Scale Faults. Large-scale faults are faults that affect up to 32 bits in the burst. Such faults may be due to full-, half-, and quarter-device faults, all row-related faults, single-bank faults, and single-lane faults.

We compute the expected probability for each class by summing the percentages of the faults. The distribution suggests that 55% of faults are single-bit faults, 4% are multi-bit faults, 4% are subsequent faults, and 37% are large-scale faults.

As we base our fault model on DRAM faults encountered in a production environment, we consider naturally occurring faults to be the main target of the error correction procedure. Rowhammer attacks [30] induce targeted bit flips in DRAM. As shown by Cojocar et al. [10], Rowhammer can bypass and exploit error-correcting codes. Voodoo would also be able to correct Rowhammer-induced errors as long as they are confined to a single block. We expect Voodoo to offer stronger protection than existing linear codes, as they can be reverse engineered and exploited [10]. However, we consider Rowhammer and hardware fault attacks out of scope as they are best mitigated at their source [41, 42].

Table 6: The fault classes that we consider in our analysis. Fault classes are ranked by the maximum number of affected bits per burst. The rate describes the percentage of faults that fall in the given class.

Class	Combined Faults	Bits	Rate
Single-bit	Single-bit	1	55%
Multi-bit	Single-word, Single-column	4	4%
Subsequent	Two-column, Single-Pin	8	4%
Large-scale	Row, Single-bank, Single-lane, Device	32	37%

5.2 Detection and Correction Capabilities

Real-world DRAM errors may violate the assumption that errors are confined to a single ciphertext block. Thus, it is essential to analyze the behavior of Voodoo under such faults.

Detection of Multi-Block Faults. A requirement for our error correction is that only one ciphertext block out of all n input blocks is faulty. However, it is necessary to reason about the behavior of Voodoo in the presence of faults spread over multiple blocks.

Assume the case in which two or more ciphertext blocks experience an error simultaneously. The syndrome will be the sum of all faults multiplied by their respective power of H. During the error correction procedure, we compute n error location indicators. Two critical events can cause data corruption. First, the syndrome may evaluate to 0 as the combined errors cancel each other out. Given that the possible range of products between errors and hash key exponents is $\{0,1\}^N \setminus \{x : \mathrm{HW}(x) \leq T_{th}\}$, the probability of this case is $2^{T_{th}-N}$. Data corruption can also occur when exactly one error location indicator suggests a correctable error. Aliasing occurs when $\mathrm{HW}\left(S \cdot H^{-j}\right) \leq T_{th}$ is fulfilled. The probability for aliasing is, thus, equal to the misinterpretation probability of the f-Unbounded encoding. Table 1 shows how the aliasing probability increases with an increasing number of correctable bits. This type of aliasing is unlikely to occur for a configuration with a block size of 128 bits. Even if $T_{th} = 14$ bits, the theoretical maximum for the given N, the misinterpretation probability is 2^{-65} (cf. Equation (8)). Thus, multi-block errors are likely to be correctly reported as uncorrectable errors.

Error Correction Capabilities. According to the fault model introduced in Section 5, the majority of faults are single-bit

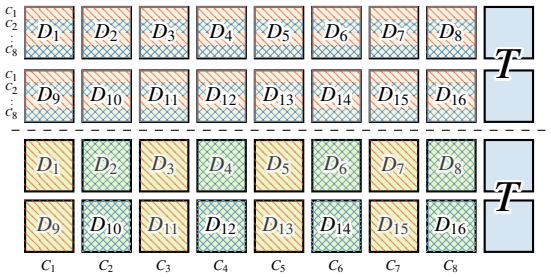

Figure 4: Two variants of how ciphertext blocks are placed in DRAM. Our approach uses the second variant (below).

faults. Every instantiation of Voodoo can handle this fault type, as T_{th} is at least 1. When considering multi-bit faults, T_{th} must be at least 4 to handle all possible faults. The configuration with $N = 64$ bits is not guaranteed to find a suitable hash key H such that Equation (6) is fulfilled for all possible errors. Thus, if we implement the selection of H as described for MAGIC, we are not guaranteed to correct subsequent faults. Even worse, it is improbable that there exists any H that would allow us to correct the large-scale faults as defined in Section 5. However, we find that we can improve the error detection and correction procedure such that some of the subsequent faults and, more importantly, large-scale faults can be detected and corrected.

Bounding Errors to Blocks. Due to Equation (6), we must assume that it is infeasible to search for a H that allows for the guaranteed correction of arbitrary errors with Hamming weights above 7 or 14 bits, depending on N. We can, however, exploit the fact that erroneous bits are usually not arbitrarily distributed. Instead, the physical structure and layout of the memory cause them to follow specific patterns. It is, for example, rather unlikely that two DRAM chips experience a fault at the same time [5, 53]. Thus, we arrange the ciphertext blocks so that one block is confined to two x4 DRAM chips for $N = 64$ bits and four x4 chips for $N = 128$ bits. This ensures that only one ciphertext block is affected even if a complete chip failure occurs.

Figure 4 illustrates two variants of how the ciphertext blocks can be distributed over the DRAM chips. In the first configuration, a single beat transmits a complete ciphertext block. However, the data of a ciphertext is distributed over all DRAM chips. When a chip experiences a subsequent or large-scale fault, multiple ciphertext blocks are affected.

The second configuration solves this by ensuring that even a complete chip fault is bounded to one ciphertext block. Thus, we only experience a multi-block fault if multiple DRAM chips in different blocks experience a fault simultaneously.

Stuck-Pin Correction. Considering the physical structure of DRAM chips for error correction allows us to extend the error detection and correction capabilities. Assume that a chip experiences a single stuck pin. As each DRAM chip is accessed eight times, the maximum number of influenced bits is 8. Thus, a maximum of 8 erroneous bits is found in a single block. For $N = 64$ bits, it is not guaranteed that we find a H with which we can correct 8-bit errors as the maximum T_{th} is limited to 7. However, the eight erroneous bits of a single pin fault are not randomly distributed. Due to the chip's structure, each faulted bit is equally spaced over the 32 bits read from the chip with the broken pin. Hence, instead of increasing T_{th} to 8 bits, it is sufficient to consider $4 \cdot (2^8 - 1) \approx 2^{10}$ error values. Note that a subset of these errors is already contained in the initially considered errors with a Hamming weight less than or equal to T_{th}. For $T_{th} = 7$ bits, there are only four errors where all 8 bits are affected that need to be considered in addition to what is covered by the regular error correction. For two stuck pins, we have $\binom{4}{2}\left(2^8 - 1\right)^2 \approx 2^{18}$ errors that need to be considered. For comparison, the number of errors that must be considered when $N = 64$ and $T_{th} = 7$ is approximately 2^{29}. Let $\mathcal{E}_{SP}$ denote the set of additional errors due to stuck pin faults. The overall set of possible errors is then $\mathcal{E} = \mathcal{E}_{SP} \cup \{e \in \{0,1\}^N : \mathrm{HW}(e) \leq T_{th}\}$. We can adopt Equation (3) to include all possible errors, thus yielding

$$|\mathcal{H}| \geq 2^N - \frac{n(n+1)}{2}(\mathcal{E})^2 .$$

With this estimation, we can determine if there exists a H that is guaranteed to fulfill Equation (6) for all considered errors. We find no significant reduction of the cardinality of $\mathcal{H}$, even when the errors stemming from single- and double-pin faults are included.

Permanent Fault Correction. For larger faults that corrupt even more bits, it is impossible to use a pattern-based approach to include them in $\mathcal{E}$. We can, however, reach Chipkill-level protection for permanent faults. According to Beigi et al. [5], most large-scale faults are permanent or intermittent. Reading from a location with a permanent fault always yields erroneous data. When performing error correction, we first identify the block that needs to be corrected. Initially, this is done by searching for an error with low Hamming weight. However, we can locate a permanently faulted chip without any computation. It is sufficient to perform two additional write and read operations to check if a chip is faulty. We can identify all permanent faults that affect the cache line by writing and reading back a known pattern and the corresponding anti-pattern. Once the faulted block is identified, we can restore the original value, just like in the case of regular error correction.

5.3 Monte-Carlo Simulation

We build a simulation showcasing the error correction and detection performance of Voodoo. Two configurations simulate the fault model defined in Section 5.1.

In the first configuration, all faults are confined to a single chip in one ciphertext block. This mode measures the percentage of correctable, miscorrected, and uncorrectable faults.

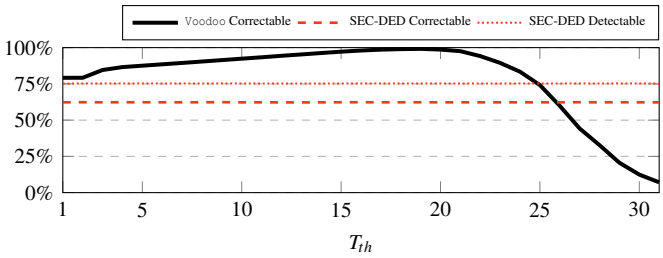

Figure 5: The percentage of correctable errors w.r.t. the threshold T_{th} assuming that a *single block* experiences faults. After an optimum at $T_{th} = 19$, the rate declines drastically.

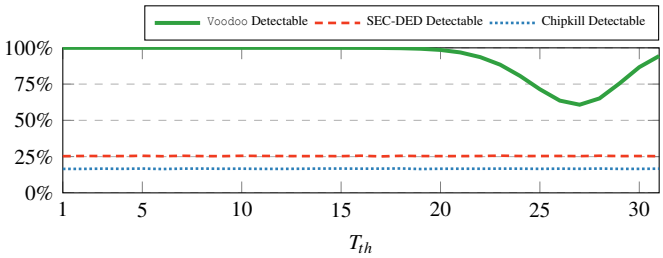

Figure 6: The percentage of detectable errors for faults in *multiple blocks*. Voodoo offers almost 100% error detection.

We compare our results to SEC-DED ECC. With SEC-DED, we consider an error correctable if only one faulted bit per ECC granule is detected. Furthermore, we consider errors as guaranteed SEC-DED detectable if the Hamming weight of the error in the ECC granule does not exceed two. While the ciphertext is confined to two chips, we assume the linear ECC checksum is computed over all 16 chips. Thus, SEC-DED can also correct errors due to single stuck pins and other errors that only affect one bit per granule. We model the distribution of faults according to [5].

The second configuration generates multi-block faults exceeding the corrective capabilities of Voodoo. This configuration measures the detection rate for multi-block faults. We measure detected and miscorrected faults. Our simulation always injects errors in two or more ciphertext blocks. Thus, none of the simulated errors are correctable. The results are compared to SEC-DED ECC and Chipkill. We assume a single-symbol-correction, double-symbol-detection Chipkill code. Hence, an error is considered Chipkill detectable if at most two chips are affected.

Both configurations use a block size of $N = 64$ bits. Furthermore, we assume a cache line size of 512 bits, resulting in $n = 8$ blocks. Each simulated access is erroneous to speed up the simulation. We draw 200 unique values for H and simulate 10 000 errors for each H. According to the data reported by Beigi et al. [5], this approximates 2^{44} years of continuous DRAM operation.

Figure 5 illustrates the percentage of correctable errors w.r.t. the selected threshold value T_{th}. Note that the percentage of correctable errors increases even when exceeding the theoretical maximum of $T_{th} = 7$. This behavior suggests that one error aliasing to another correctable error is very unlikely, even with a high T_{th}. Hence, miscorrections are also extremely rare. We are far more likely to deem an error uncorrectable because the error location is not uniquely identifiable. We find that small values of T_{th} already allow us to surpass the corrective capabilities of SEC-DED ECC. With Voodoo, every error confined to a single block is guaranteed to be detected.

Figure 6 illustrates the detection rate of Voodoo for multi-block errors. Voodoo correctly identifies nearly all multi-block errors as uncorrectable for low values of T_{th}. For $T_{th} = 27$ the miscorrection rate reaches its maximum at 39%. At this optimum, the aliasing probability from multi-block errors to single-block errors is the highest. Further increasing T_{th} will decrease the miscorrection rate, as the likelihood that a multi-block error aliases to two or more single-block errors increases. This behavior is also reflected by Equation (8), which has an optimum at $T_{th} = 27$ bits.

In summary, our Monte-Carlo simulation results underline the applicability of Voodoo for error detection and correction. As expected, the threshold T_{th} strongly influences the corrective capabilities of Voodoo. When setting T_{th} appropriately, we can correct almost all errors produced by the fault model defined in Section 5.1. This strong correction performance is due to most errors being limited in the number of erroneous bits, and the rarity of large-scale errors affecting many bits.

5.4 Error Correction Latency

As Voodoo performs a fixed number of steps on each error correction, we can estimate the time needed for a single correction. The correction latency only depends on the time it takes to correctly identify the faulted block. In a configuration with n N-bit blocks, the number of error location indicators equals n. These error location indicators are computed as $\forall i \in [1,n] : S_i = S \cdot H^{-i}$ and the error correction succeeds if $HW(S_i) \leq T_{th}$ holds for exactly one location indicator. We can consider two approaches for this search. First, we can perform the computation and the check iteratively, resulting in n subsequent computations. Assuming a 0.44 ns latency for each multiplication [3] and $n = 8$ ciphertext blocks, the overall search latency equals 3.52 ns. Contrarily, a fully parallelized search only takes 0.44 ns. Before locating the error, Voodoo decrypts the stored checksum, thus imposing an additional latency of 2.2 ns when using QARMA.

Latency Comparison. We compare our correction latency with the latencies of commodity ECC, HashTag [33], and MUSE ECC [40]. This selection of schemes allows us to gauge the performance of Voodoo compared to commodity error correction, iterative error correction, and correction based on residual codes. According to Cojocar et al. [10] and

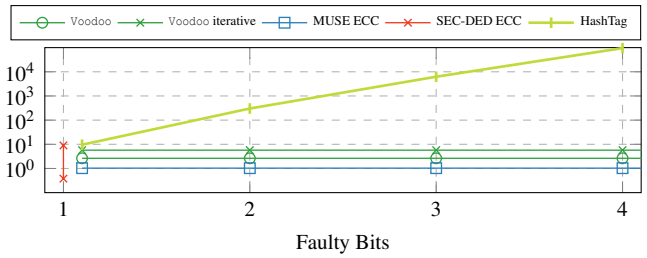

Figure 7: The error correction latencies of different error detection and correction schemes in ns.

Kwong et al. [32], the correction latencies of ECC DDR4 DRAM can range from 0.9 ns to 9 ms. Manzhosov et al. [40] report a correction latency of 0.38 ns for an RS(144,128) code. We use this latency as the lower bound of commodity error correction. For HashTag, we assume that errors are confined to a single 64-bit block and the search is limited to the affected block. We assume that HashTag uses SPEEDY with a latency of 0.3 ns per invocation. For MUSE ECC, we consider a configuration with Chipkill error correction. The error detection and correction latencies of MUSE ECC range from 0.856 ns to 1.179 ns, with the average being 1.03 ns.

Figure 7 shows the error correction latency of Voodoo, MUSE ECC, HashTag, and a commodity ECC solution in direct comparison. The latency of Voodoo is reasonable low at 2.64 ns and 5.72 ns for the parallel and the iterative approach, respectively. This places our latency within the reported range of latencies for commodity ECC [10, 32, 33]. MUSE ECC outperforms Voodoo in terms of correction latency. Note that the main latency of error correction with parallel Voodoo comes from the decryption of the stored checksum. Using a faster block cipher will, thus, result in a lower error correction latency. Both implementations of Voodoo's search outperform the error correction of HashTag. Furthermore, the latency of HashTag grows as the number of faulty bits increases while Voodoo and MUSE ECC offer error correction in a fixed amount of time.

6 Evaluation

In this section, we evaluate Voodoo. While the Monte-Carlo simulation presented in Section 5 allows us to evaluate the error correction performance, we also implement Voodoo using the gem5 full-system simulator [7] to measure the overall performance impact. For comparison, we implement an MTE-like tagged memory architecture that is implemented independently from the authenticated encryption and error correction provided by Voodoo.

6.1 Implementation in gem5

We implement our proof-of-concept prototype using the gem5 system simulator (version 22.1.0.0) to provide an accurate hardware model of our design. Additionally, we implement an authenticated encryption engine based on the structure shown in Figure 2. We implement an MTE-like tagged memory architecture to measure the performance overheads of traditional memory tagging.

Voodoo Mode. We implement Voodoo as described in Section 3.1 as an extension to the gem5 simulator. We include a module with configurable encryption and decryption latencies in our implementation. This allows us to parameterize the model according to the selection of the underlying block ciphers. This model measures the overheads generated by adding the encryption layer and the logic for the MAC computation. We assume that instances of the QARMA block cipher are used for encryption. According to Avanzi [3], the latency of QARMA_x-64 can range from 2.2 ns to 3.6 ns. The delay of a 64-bit GF(2^{64}) multiplication is listed at 0.44 ns. Leander et al. [35], however, list their minimum achieved latency for QARMA_x-64 at approximately 0.4 ns. We make a conservative estimation of the combined forward and backward latencies that we impose on each memory request. When writing to memory, we add a forward latency of 4.04 ns to each write access. This latency is the estimated time it will take to compute the MAC for the provided data. When reading from memory, we impose a 3.6 ns delay, as the checksum is decrypted while the fresh checksum is computed.

Memory Tagging. We implement a memory tagging model that integrates a tagged memory architecture for storing and propagating tags in hardware. We co-locate the tag metadata with the data for every cache line. Additional memory requests are required for every cache miss to fetch the associated tag metadata from a dedicated DRAM region. This mode represents a naïve tagged memory implementation.

Authenticated Encryption and Memory Tagging. This mode implements memory tagging and authenticated encryption as two disjunct system features. Each memory access is delayed, as described above. Additionally, memory accesses trigger additional loads from DRAM.

6.2 Performance Evaluation

We evaluate our hardware models using gem5 and the SPEC CPU2017 benchmark suite running on Linux v5.15. Similar to existing work [33, 40, 58, 60] and to avoid infeasible simulation times, we use the TimingSimpleCPU model at 3 GHz. The cache hierarchy is configured to a private 8-way set associative L1i and L1d cache with 16 kB and 64 kB, respectively. The L1 and L2 access latency is configured to 1 and 10 cycles, respectively. We use a 8 MB shared 16-way set-associative L2 and an 8 GB DDR4 DRAM module.

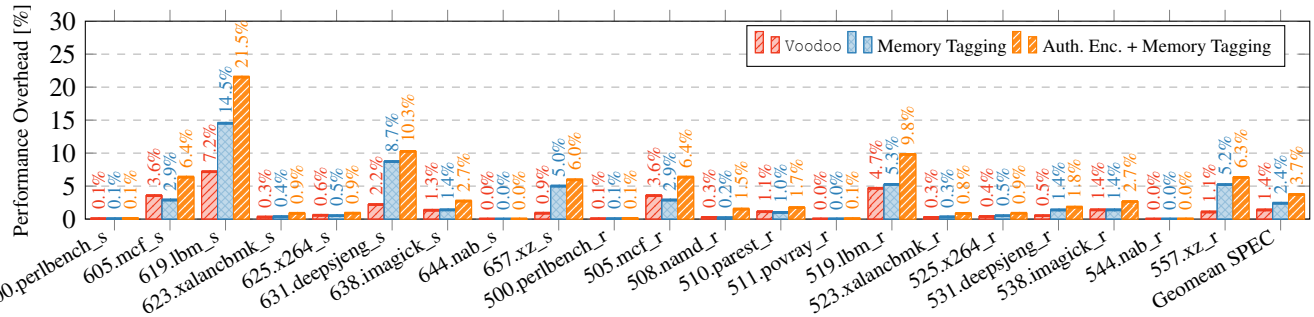

Figure 8: Simulated performance overhead using the SPEC CPU2017 benchmark suite.

Simulation Results. Figure 8 illustrates the simulated performance overhead. The baseline performance is determined by measuring the runtime of the benchmarks on a system that does not implement authenticated encryption or memory tagging. We excluded the benchmarks that encounter toolchain issues, compilation failures, or runtime errors.

The simulated performance results of our design show an overhead of 1.4% compared to the baseline system. The overhead is caused by the delays due to the memory encryption. For traditional memory tagging, we measure an average performance overhead of 2.4%. The additional DRAM requests cause this overhead and it increases for benchmarks with high memory pressure. for the authenticated encryption with separate memory tagging, we find that the geomean of the performance overhead is 3.7%.

Performance Comparison. We compare the performance of `Voodoo` to the reported performance numbers of commodity memory encryption schemes and a commodity tagged memory implementation. We compare `Voodoo` against AMD's memory encryption engine, the memory encryption of Intel TDX, and ARM's MTE implementation. AMD's memory encryption engine uses an optimized AES implementation and imposes a reported geomean performance overhead of 1% across workloads [34]. New and upcoming Intel CPUs incorporate a memory encrpytion engine using AES-XTS with 128-bit keys. The reported maximum performance overhead of Intel TME-MK is given at 2.2% [21]. For ARM MTE, a recent study reports a geomean performance overhead of 1.5% for the SPEC CPU2006 benchmark suite [16]. These results highlight the competitive performance of `Voodoo` compared to other implementations of memory encryption and tagging. Note that the performance given for `Voodoo` in Figure 8 is achieved with an in-order CPU model while the reported performance numbers given above stem from commodity hardware.

6.3 Area Estimation

We estimate the area overhead using the available data for QARMA presented by Avanzi [3]. The area overhead of a single multiplication in $GF(2^{64})$ is listed at 17 kGE ($924.6\,\mu m^2$). Assuming an unoptimized implementation with $n = 8$ message blocks of $N = 64$ bits each, we end up with 8 instances of the 64-bit multipliers, summing up to 136 kGE ($7397\,\mu m^2$). On top of the multiplication, we must also account for the block cipher instances. The choice of the block cipher has a strong impact on the overall area overhead. Assuming a fully-unrolled QARMA-64 with 22kGE ($1238\,\mu m^2$) as the underlying block cipher and no further optimizations, we require 9 instances of the block cipher, resulting in 198kGE ($11\,142\,\mu m^2$). Thus, we reach an overall area overhead of approximately 220kGE ($18\,539\,\mu m^2$) for `Voodoo`. Using the same comparison as [33], we reach a relative overhead of 0.007% when considering the area of a Raptor Lake CPU [57].

7 Related Work

Several academic designs propose using ECC DRAM modules to encode metadata or increase system performance for tagged architectures. However, none of the related research proposes a combined primitive for authenticated encryption, error correction, and memory tagging.

HashTag [33] replaces Hamming codes with truncated hashes and stores the memory tags in free bits of the ECC chip. While this approach eliminates additional fetches, the error correction procedure suffers from increased latencies due to a brute-force search. Furthermore, large tag sizes reduce the error detection and correction capabilities significantly. `Voodoo` offers a larger tag space without compromising on error correction. Furthermore, our error correction approach does not rely on an iterative search.

MUSE ECC [40] uses residue codes to protect DRAM integrity. The unused states of the code provide spare bits that can be used for memory tagging. The number of spare bits depends on the size of the code and the error detection and correction capabilities. Contrary to `Voodoo`, MUSE ECC only protects data integrity.

Implicit Memory Tagging (IMT) [55] uses alias-free tagged ECC for memory tagging, where they integrate the tag metadata within the check bits of the ECC chip. IMT detects tag

mismatches while maintaining single-bit error correction. It is, however, impossible to extract a tag encoded into the tagged ECC. Thus, tag-read operations are not supported.

SYNERGY [47] uses ECC DRAM to co-locate MACs from authenticated encryption with data, effectively removing additional DRAM requests and providing error correction through externally stored redundancy. Thus, SYNERGY does not reduce the memory overhead but eliminates the performance overhead of MAC fetches in the error-free case.

8 Conclusion

In this paper, we present Voodoo, the first combined scheme for authenticated encryption, DRAM integrity protection, and memory tagging. We extend MAGIC and develop three novel encodings for memory tags to allow for memory tagging without additional memory requests or storage requirements. We thus address the limitations of conventional tagged architectures, which incur non-negligible performance and memory overheads. We showcase that a wide variety of existing tagged memory architectures can be realized using our design. Moreover, we analyze our scheme's robustness based on real-life DRAM fault data. Our gem5 prototype implementation demonstrates the advantages of our approach when compared to a conventional tagged memory implementation. Compared to a system without any additional security features, our Voodoo prototype incurs a low performance overhead of 1.4% on average. In addition, our Monte-Carlo simulation shows the advantages of Voodoo over SEC-DED codes as we reach an error correction rate of 99% for single-chip errors. Furthermore, we reach a higher detection rate for multi-block faults than Chipkill.

Acknowledgments

We thank the anonymous reviewers and our shepherd for their valuable feedback that greatly improved this work. This project has received funding from the Austrian Research Promotion Agency (FFG) via the SEIZE project (FFG grant number 888087) and the AWARE project (FFG grant number 891092).

References

[1] Kathirgamar Aingaran, Sumti Jairath, Georgios K. Konstadinidis, Serena Leung, Paul Loewenstein, Curtis McAllister, Stephen Phillips, Zoran Radovic, Ram Sivaramakrishnan, David Smentek, and Thomas Wicki. M7: Oracle's Next-Generation Sparc Processor. *IEEE Micro*, 2015.

[2] AMD. Strengthening VM isolation with integrity protection and more. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/solution-briefs/amd-secure-encrypted-virtualization-solution-brief.pdf, 2020. Accessed: 2023-07-02.

[3] Roberto Avanzi. The QARMA Block Cipher Family - Almost MDS Matrices Over Rings With Zero Divisors, Nearly Symmetric Even-Mansour Constructions With Non-Involutory Central Rounds, and Search Heuristics for Low-Latency S-Boxes. *IACR Cryptol. ePrint Arch.*, 2016.

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

[5] Majed Valad Beigi, Yi Cao, Sudhanva Gurumurthi, Charles Recchia, Andrew C. Walton, and Vilas Sridharan. A Systematic Study of DDR4 DRAM Faults in the Field. In *HPCA*, 2023.

[6] Bruno Bierbaumer, Julian Kirsch, Thomas Kittel, Aurélien Francillon, and Apostolis Zarras. Smashing the Stack Protector for Fun and Profit. In *SEC*, 2018.

[7] Nathan L. Binkert, Bradford M. Beckmann, Gabriel Black, Steven K. Reinhardt, Ali G. Saidi, Arkaprava Basu, Joel Hestness, Derek Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib Bin Altaf, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *SIGARCH Comput. Archit. News*, 2011.

[8] Nicholas P. Carter, Stephen W. Keckler, and William J. Dally. Hardware Support for Fast Capability-based Addressing. In *ASPLOS*, 1994.

[9] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N. Asokan, and Danfeng (Daphne) Yao. Exploitation Techniques for Data-oriented Attacks with Existing and Potential Defense Approaches. *ACM Trans. Priv. Secur.*, 2021.

[10] Lucian Cojocar, Kaveh Razavi, Cristiano Giuffrida, and Herbert Bos. Exploiting Correcting Codes: On the Effectiveness of ECC Memory Against Rowhammer Attacks. In *S&P*, 2019.

[11] Intel Corporation. Intel Trust Domain Extensions (Intel TDX). https://www.intel.com/content/www/us/en/developer/articles/technical/intel-trust-domain-extensions.html, 2020. Accessed: 2022-09-01.

[12] Victor Costan and Srinivas Devadas. Intel SGX Explained. *IACR Cryptol. ePrint Arch.*, 2016.

[13] Jedidiah R. Crandall, Shyhtsun Felix Wu, and Frederic T. Chong. Minos: Architectural support for protecting control data. *ACM Trans. Archit. Code Optim.*, 2006.

[14] 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.

[15] Morris Dworkin. SHA-3 Standard: Permutation-Based Hash and Extendable-Output Functions, 2015-08-04 2015.

[16] Floris Gorter, Taddeus Kroes, Herbert Bos, and Cristiano Giuffrida. Sticky Tags: Efficient and Deterministic Spatial Memory Error Mitigation using Persistent Memory Tags. In *S&P*, May 2024.

[17] Shay Gueron. Memory Encryption for General-Purpose Processors. *IEEE Secur. Priv.*, 2016.

[18] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *USENIX*, 2008.

[19] Richard W Hamming. Error detecting and error correcting codes. *The Bell system technical journal*, 29(2):147–160, 1950.

[20] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks. In *S&P*, 2016.

[21] Intel. Runtime Encryption of Memory with Intel® Total Memory Encryption–Multi-Key (Intel® TME-MK). https://www.intel.com/content/www/us/en/developer/articles/news/runtime-encryption-of-memory-with-intel-tme-mk.html. Accessed: 2024-02-16.

[22] Intel. Intel Architecture Memory Encryption Technologies. https://www.intel.com/content/www/us/en/content-details/679154/intel-architecture-memory-encryption-technologies-specification.html, 2022. Revision 1.4, Accessed: 2023-01-31.

[23] Kyriakos K. Ispoglou, Bader AlBassam, Trent Jaeger, and Mathias Payer. Block Oriented Programming: Automating Data-Only Attacks. In *CCS*, 2018.

[24] Bruce L. Jacob, Spencer W. Ng, and David T. Wang. *Memory Systems: Cache, DRAM, Disk*. 2008.

[25] JEDEC. DDR4 SDRAM STANDARD. *JESD79-4, Sep*, 2012.

[26] Samuel Jero, Nathan Burow, Bryan C. Ward, Richard Skowyra, Roger Khazan, Howard E. Shrobe, and Hamed Okhravi. TAG: Tagged Architecture Guide. *ACM Comput. Surv.*, 2023.

[27] Alexandre Joannou, Jonathan Woodruff, Robert Kovacsics, Simon W. Moore, Alex Bradbury, Hongyan Xia, Robert N. M. Watson, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alfredo Mazzinghi, Alex Richardson, Stacey D. Son, and A. Theodore Markettos. Efficient Tagged Memory. In *ICCD*, 2017.

[28] Hari Kannan, Michael Dalton, and Christos Kozyrakis. Decoupling Dynamic Information Flow Tracking with a dedicated coprocessor. In *DSN*, 2009.

[29] David Kaplan, Jeremy Powell, and Tom Woller. AMD Memory Encryption. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf, 2021. Accessed: 2023-02-26.

[30] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ISCA*, 2014.

[31] Michael E. Kounavis, David Durham, Sergej Deutsch, Krystian Matusiewicz, and David Wheeler. The MAGIC Mode for Simultaneously Supporting Encryption, Message Authentication and Error Correction. *IACR Cryptol. ePrint Arch.*, 2020.

[32] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading Bits in Memory Without Accessing Them. In *S&P*, 2020.

[33] Lukas Lamster, Martin Unterguggenberger, David Schrammel, and Stefan Mangard. HashTag: Hash-based Integrity Protection for Tagged Architectures. In *USENIX*, 2023.

[34] Michael Larabel. AMD Secure Memory Encryption "SME" Performance With 4th Gen EPYC Genoa. https://www.phoronix.com/review/amd-sme-genoa/5, 2022. Accessed: 2024-02-16.

[35] Gregor Leander, Thorben Moos, Amir Moradi, and Shahram Rasoolzadeh. 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.

[36] Scott Levy, Kurt B. Ferreira, Nathan DeBardeleben, Taniya Siddiqua, Vilas Sridharan, and Elisabeth Baseman. Lessons learned from memory errors observed over the lifetime of Cielo. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018*, 2018.

[37] Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. CrossLine: Breaking "Security-by-Crash" based Memory Isolation in AMD SEV. In *CCS*, 2021.

[38] Jamie Liu, Ben Jaiyen, Yoongu Kim, Chris Wilkerson, and Onur Mutlu. An experimental study of data retention behavior in modern DRAM devices: implications for retention time profiling mechanisms. In *ISCA*, 2013.

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

[40] Evgeny Manzhosov, Adam Hastings, Meghna Pancholi, Ryan Piersma, Mohamed Tarek Ibn Ziad, and Simha Sethumadhavan. Revisiting Residue Codes for Modern Memories. In *MICRO*, 2022.

[41] Michele Marazzi, Patrick Jattke, Flavien Solt, and Kaveh Razavi. ProTRR: Principled yet Optimal In-DRAM Target Row Refresh. In *S&P*, 2022.

[42] Michele Marazzi, Flavien Solt, Patrick Jattke, Kubo Takashi, and Kaveh Razavi. REGA: Scalable Rowhammer Mitigation with Refresh-Generating Activations. In *S&P*, 2023.

[43] David McGrew and John Viega. The Galois/counter mode of operation (GCM). *submission to NIST Modes of Operation Process*, 20:0278–0070, 2004.

[44] Arjun Menon, Subadra Murugan, Chester Rebeiro, Neel Gala, and Kamakoti Veezhinathan. Shakti-T: A RISC-V Processor with Light Weight Security Extensions. In *HASP*, 2017.

[45] Microsoft. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. https://github.com/Microsoft/MSRC-Security-Research/blob/master/presentations/2019_02_BlueHatIL/2019_01%20-%20BlueHatIL%20-%20Trends%2C%20challenge%2C%20and%20shifts%20in%20software%20vulnerability%20mitigation.pdf, 2019. Accessed: 2023-02-26.

[46] Phillip J Restle, JW Park, and Brian F Lloyd. DRAM variable retention time. *IEDM Tech. Dig*, pages 807–810, 1992.

[47] Gururaj Saileshwar, Prashant J. Nair, Prakash Ramrakhyani, Wendy Elsasser, and Moinuddin K. Qureshi. SYNERGY: Rethinking Secure-Memory Design for Error-Correcting Memories. In *HPCA*, 2018.

[48] David Schrammel, Moritz Waser, Lukas Lamster, Martin Unterguggenberger, and Stefan Mangard. SPEAR-V: Secure and Practical Enclave Architecture for RISC-V. In *ASIACCS*, 2023.

[49] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS*, 2009.

[50] Kostya Serebryany. ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety. *login Usenix Mag.*, 2019.

[51] Kostya Serebryany, Evgenii Stepanov, Aleksey Shlyapnikov, Vlad Tsyrklevich, and Dmitry Vyukov. Memory Tagging and how it improves C/C++ memory safety. *CoRR*, 2018.

[52] Chengyu Song, Hyungon Moon, Monjur Alam, Insu Yun, Byoungyoung Lee, Taesoo Kim, Wenke Lee, and Yunheung Paek. HDFI: Hardware-Assisted Data-Flow Isolation. In *S&P*, 2016.

[53] Vilas Sridharan, Nathan DeBardeleben, Sean Blanchard, Kurt B. Ferreira, Jon Stearley, John Shalf, and Sudhanva Gurumurthi. Memory Errors in Modern Systems: The Good, The Bad, and The Ugly. In *ASPLOS*, 2015.

[54] G. Edward Suh, Jae W. Lee, David Zhang, and Srinivas Devadas. Secure program execution via dynamic information flow tracking. In *ASPLOS*, 2004.

[55] Michael B. Sullivan, Mohamed Tarek Ibn Ziad, Aamer Jaleel, and Stephen W. Keckler. Implicit Memory Tagging: No-Overhead Memory Safety Using Alias-Free Tagged ECC. In *ISCA*, 2023.

[56] Laszlo Szekeres, Mathias Payer, Tao Wei, and Dawn Song. SoK: Eternal War in Memory. In *S&P*, 2013.

[57] 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.

[58] Martin Unterguggenberger, David Schrammel, Pascal Nasahl, Robert Schilling, Lukas Lamster, and Stefan Mangard. Multi-Tag: A Hardware-Software Co-Design for Memory Safety based on Multi-Granular Memory Tagging. In *ASIACCS*, 2023.

[59] Samuel Weiser, Mario Werner, Ferdinand Brasser, Maja Malenko, Stefan Mangard, and Ahmad-Reza Sadeghi. TIMBER-V: Tag-Isolated Memory Bringing Fine-grained Enclaves to RISC-V. In *NDSS*, 2019.

[60] Mario Werner, Thomas Unterluggauer, Lukas Giner, Michael Schwarz, Daniel Gruss, and Stefan Mangard. ScatterCache: Thwarting Cache Attacks via Cache Set Randomization. In *USENIX*, 2019.

[61] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. SEVurity: No Security Without

Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *S&P*, 2020.

[62] Emmett Witchel, Josh Cates, and Krste Asanovic. Mondrian memory protection. In *ASPLOS*, 2002.

[63] Jonathan Woodruff, Robert N. M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert M. Norton, and Michael Roe. The CHERI capability model: Revisiting RISC in an age of risk. In *ISCA*, 2014.

[64] Nickolai Zeldovich, Hari Kannan, Michael Dalton, and Christos Kozyrakis. Hardware Enforcement of Application Security Policies Using Tagged Memory. In *OSDI*, 2008.

A Appendix

A.1 DRAM Error Simulation

Table 7 lists the fault types given in Table 6 in more detail. As the study by Beigi et al. measures fault rates for two different DRAM vendors, we average over the reported rates. Our Monte-Carlo simulation samples from the given set of faults. After deciding the fault type, we generate the number of faulted bits depending on the fault type. For single-word and single-column faults, we uniformly draw a number between 1 and 4 deciding how many bits are faulted. We discard zero-bit results. We then sample an error that has the previously sampled number of bits set. For larger faults such as row faults, we draw the number of faulted columns according to the fault type. We then proceed to draw the number of faulted bits per faulted column. The number of faulted columns in the same burst window is decided by computing the probability that we see only one faulted column in the burst and sampling from this distribution. If multiple columns are present, we draw the number of affected columns uniformly and allow for up to 8 faulted columns in the burst window. Thus, row faults that affect multiple columns are slightly exaggerated, as the number of faulted columns per burst window would follow a hypergeometric distribution.

A.2 Glossary of Terms

Table 8 lists the symbols and terms used in this paper.

Table 7: The detailed fault classes that we consider in our analysis.

Class	Fault	Bits	Rate
Single-bit	Single-bit	1	55.06%
Multi-bit	Single-word	4	0.325%
	Single-column	4	3.85%
Subsequent	Two-column	8	2.84%
	Single-pin	8	0.67%
Large-scale	Partial row	32	24.345%
	Single row	32	0.26%
	Single row + single bit	32	0.975%
	Two row	32	4.125%
	Consecutive row	32	0.555%
	Cluster row	32	5.7%
	Single bank	32	0.065%
	Quarter device	32	0.135%
	Half device	32	0.09%
	Full device	32	0.605%
	Single lane	32	0.4%

Table 8: Table of symbols

n	Number of message blocks
X	Tag Size
N	Ciphertext Block Size
P	Probability
S	Syndrome (Equation (5))
T	Checksum (Equations (1), (4) and (10))
e_j	Error in j-th block
e_T	Error between stored and supplied memory tag
HW	Hamming Weight
T_{th}	Hamming weight threshold for error correction
L_L, L_U	Hamming weight bounds for f-Encrypt
M_j	Message block j
M_T	Encoded Memory Tag
M'_T	Encoded Memory Tag supplied during the memory access
C_i	Ciphertext block i
C'_i	Ciphertext block i read from DRAM
H	Secret hash key
H^i	i-th exponent of hash key
$\mathcal{H}$	Set of feasible hash keys (Equation (3))
$\mathcal{C}_f$	Set of misinterpreted error location indicators (Equation (7))
$\mathcal{M}_\mathcal{T}$	Set of tags for f-Bounded (Equation (12))
E_D	Data encryption cipher
E_K	Blinding cipher for MAC