The Impact and Detection of Silent Data Corruption on Erasure Coding

Jiyu Hu
Carnegie Mellon University
Pittsburgh, USA
jiyuh@andrew.cmu.edu

Yiwei Zhao
Carnegie Mellon University
Pittsburgh, USA
yiweiz3@andrew.cmu.edu

Yiyang Guo
Carnegie Mellon University
Pittsburgh, USA
yiyangg@andrew.cmu.edu

ABSTRACT

There is a growing concern about silent data corruption in large-scale datacenters. Silent data corruptions in datacenters not only significantly impact data integrity but also require a huge amount of effort and resource to detect and prevent. In this project, we aim to study the influence of silent data corruption on erasure coding, a widely adopted redundancy method from the domain of coding theory. Understanding the impact of silent data corruption on erasure coding is extremely important because various distributed systems, especially storage systems, rely on the integrity of erasure codes to prevent data loss. Moreover, we believe that the computation of erasure coding can potentially serve as an silent data corruption detection routine since it is regularly performed in the systems.

This work provides an initial modeling of the propagation of silent data corruption in erasure coding applications. An error injection tool has been implemented on an existing erasure coding library. Additionally, we provide a strawman detection of silent data corruption oblivious to encoding methods, as well as encoding-aware detection methods for scheduling-based encoding. Our evaluation shows our detection brings no more than 20% and 10% respective overhead in encoding to detect all possible errors.

CCS CONCEPTS

• **Hardware → Error detection and error correction**; • **Computer systems organization → Reliability**; **Redundancy**.

KEYWORDS

erasure code, silent data corruption, error detection

1 INTRODUCTION

Storage systems utilize redundancy to guarantee data integrity in the case of failures. *Erasure coding* (EC), a special type of *error correction codes* (ECC), is a leading method that enables storage systems to store data reliably but with significantly less storage overhead than replication [15]. Due to these properties, EC methods are widely used in state-of-the-art storage systems, varying from hard disk [3] and solid state drive (SSD) arrays [11] to distributed [5] and cloud storages [8].

However, the reliability of EC is increasingly threatened by the existence of *silent data corruption* (SDC). SDC is an error type that gives out corrupted computation results (e.g., $2 + 2 = 5$) without notifying the upper-level application that corrupted data has been generated. Traditional sense of SDC is considered to be triggered by outer-space cosmic radiation that causes transistors inside the electronic units to flip [12]. Recently, SDC has been attributed to a mix of common problems, including manufacturing defects of electronic components such as central processing units (CPU) and dynamic random-access memory (DRAM). There has been a growing concern for large-scale datacenter providers such as Facebook [4] and Google [6], where both the encoding and decoding processes of EC are prone to SDC. Consequently, the data protected by EC is no longer guaranteed to be recoverable.

SDC severely destroys the reliability of EC. One challenging in resolving the negative impacts of SDC is that the error pattern is highly dependant on the computation process. For instance, numerous methods have been proposed to accelerate the computation of EC, such as the ones based on bitmatrix [13], bitmatrix normalization [20], smart scheduling [18] and heuristic matching [7]. Even though the same resulting code is eventually produced given the same set of parameters, the computation procedures differ tremendously between various methods. Thus, a single bit-flip caused by SDC will generate completely distinct propagation patterns when it appears at different points in different computation methods.

In this project, we first want to evaluate the influence of SDC on different commonly-used EC computation algorithms in a quantitative manner. We implement an error injection tool to simulate SDC on top of the Jerasure library [20], a highly-optimized library for EC computation. We carry out experiments and model the propagation patterns of SDC in commonly used EC encoding methods, i.e., matrix-based, bitmatrix-based and smart scheduling. The error injection tool can also be extended to other EC codebases developed from Jerasure, such as Zerasure [23], which we leave the modeling of these state-of-the-art methods as future work.

In addition, we provide a potential solution to SDC, i.e., SDC detection (where computation are back-rolled when potential SDC is detected). We propose and implement two SDC detection methods – a strawman detection that is oblivious over all encoding methods, and a lighter-weight detection based on heavy path decomposition that is tailored towards smart scheduling [18]. We also propose a heuristic greedy detection for all scheduling-based methods, but have not implemented it due to the lack of baseline codebases to test on. We also leave this as future work.

In summary, this project is an initial work of SDC on EC. We hope our work can lay a foundation and shed light on future researches. Our main contributions are:

- An SDC injection tool on the Jearsure library.
- Modeling, analysis, and experiments on the error propagation patterns of SDC on methods in Jearasure.
- Three proposed SDC detection methods, with two of them implemented and evaluated for computation overhead.

The rest of the paper is organized as follows. Section 2 introduces background and related works on SDC and EC. Section 3 describes preliminaries and some basic building blocks used in our modeling and detection. Section 4 explains our analysis of error propagation

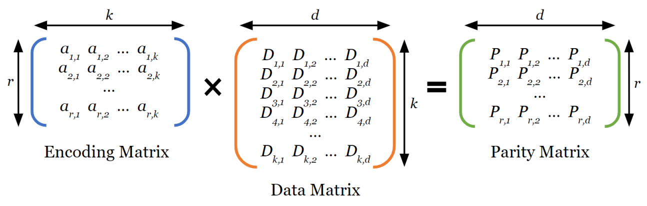


Figure 1: High-level encoding operation from [23]

patterns of SDC and plots our design of SDC detection. Section 5 describes the implementation details for both error injection and detection. We carry out experiments simulating SDC propagation in EC, and evaluate our detection design in Section 6. Eventually, we conclude in Section 7.

2 BACKGROUND AND RELATED WORKS

2.1 Silent Data Corruption

Silent Data Corruption [4], sometimes referred to as Silent Data Error (SDE), is an industry-wide issue impacting not only long-protected memory, storage, and networking, but also computer CPUs . CPUs give out corrupted computation results in the presence of SDCs, without event logging of the mistakes. Typical causes include radiation [1], synthetic fault injection [10] and device characteristics [4].

Researchers are handling the problems of SDC mainly from two angles, hardware and software. Leray et al. [12] described using hardware redundancy to cope with SDCs, adding extra circuits to detect and correct corrupted data. On the other hand, most of the recent works focus on software approaches, such as the ones from Facebook [4] and Google [6]. This is because designing and manufacturing custom hardware is often expensive and infeasible.

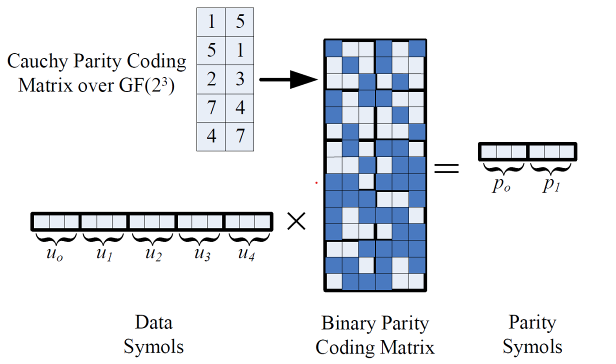
In this project, we aimed at resolving SDC from a software perspective, and we focus on dealing with the specific application case of erasure code generation.

2.2 Erasure Coding

A naive erasure code encodes k data units to produce additional r parity units to produce the combined $(k + r)$ units. Parity units are formed by a linear combination of the data units such that reading any k of the $(k + r)$ data and parity units is able to recover the original k data units. This property holds for a class of codes known as the "Maximum Distance Code" (MDS). MDS codes, such as Reed-Solomon Code [21] and Cauchy Reed-Solomon Code [13], are the most commonly used erasure coding methods in storage systems. Next, we describe the encoding and decoding process of erasure coding in detail.

Encoding. The erasure coding encoding process takes in k data units and produces r parity units. Since each parity unit is formed via a linear combination of k data units, the encoding of a single parity unit can be viewed as a dot product of a vector of k coefficients and a vector of k data elements. As shown in Figure 1, each data unit hold d elements, thus the encoding process can be viewed as a matrix multiplication of an encoding matrix G of size $(r \times k)$ and a data matrix D of size $(k \times d)$. The parity matrix, together with the data matrix, forms the $(k+r)$ units. This step is usually achieved by concatenating an identity matrix I of size (k, k) with the encoding matrix to form an extended encoding matrix $G' = [I, G]$ of size $(k + r, d)$. However, the computations performed are under finite fields (Galois Fields), which are much more costly than normal arithmetic operations. As a result, many erasure coding optimization techniques that are used in practice do not treat the calculation as matrix multiplication at all. Detailed optimizations are introduced in Section 2.3.

Decoding. To reconstruct lost data, the erasure code decoding process computes the lost data units through a similar matrix multiplication process as encoding. Assuming that r data units are lost, we first eliminate the corresponding r rows in G'. The new matrix is denoted G_e. Then we gather the k remaining data and parity units to form a matrix A. The matrices satisfy the following equation: $A = C_e D$, and then the lost data can be reconstructed by $D = {C_e}^{-1} A$.


Cauchy Parity Coding Matrix over GF(2³)
1 5
5 1
2 3
7 4
4 7
u0 u1 u2 u3 u4
p0 p1
Data Symols
Binary Parity Coding Matrix
Parity Symols
Figure 2: Encoding a $(5, 2, 3)$-code in bitmatrix representation from [23].

2.3 Optimizations for EC Computation

It is expensive to calculate erasure codes directly using finite field operations on parity matrix representation in computer systems. However, the computation can be transformed into XOR and copying operations using a bitmatrix representation. Various further optimizations on the computation have been proposed in the past few decades.

Bitmatrix Representation. In the bitmatrix representation [2], an element e in $GF(2^w)$ is represented as a $1 \times w$ row vector $V(e)$ or a $w \times w$ matrix $M(e)$, whose elements are single-bit values of 1/0 [i.e., an element in $GF(2)$]. $V(e)$ is equivalent to the bit-vector of e, while the i-th row in $M(e)$ is the bit-vector $V(e^{2^{i-1}})$. The bitmatrix representation converts each element in the original parity matrix in $GF(2^w)$ to a chunk of binary sub-matrix with size $w \times w$. Meanwhile, the multiplication in parity matrix representation is converted to XOR and copying operations in the bitmatrix representation, which is friendly to both hardware and software on modern computers. An example of this conversion is illustrated in Figure 2.

Bitmatrix Normalization. The number of XOR operations in encoding is the same as the number of 1s in the bitmatrix, so one way to reduce computation overhead in encoding is to heuristicly reduce the number of 1s in the parity bitmatrix [16].

Bitmatrix normalization [20] proposes a more systematic way to reduce this number of 1s. It is observed that multiplying each row/column by a non-zero constant does not change the invertibility of the parity matrix. Thus, one way to reduce computation is to normalize row/columns to generate more 0s in the bitmatrix representation and multiplicative unit 1 in the parity representation. To be specific, it first multiplies each row in the parity representation by normalizing the first element to be 1. Then, it normalize each column (except the first one) that generates the least number of 1s in the bitmatrix representation with the constraint that one of the element in this column is normalized to 1 in the parity representation.

Smart Scheduling. Another family of approach to reduce encoding overhead is to reuse intermediate results during the computation. Smart scheduling [18] is proposed with the design idea that more efficient computation schedules might exist if a parity bit can be written as the XOR of another parity bits and a few data bits. In the following example, $o_2^{(2)}$ calculates the same result much more efficiently than $o_2^{(1)}$.

$$o_1 = i_1 \oplus i_2 \oplus i_3 \oplus i_4 \oplus i_5$$
$$o_2^{(1)} = i_1 \oplus i_2 \oplus i_3 \oplus i_4 \oplus i_6$$
$$o_2^{(2)} = o_1 \oplus i_5 \oplus i_6$$

Smart scheduling method is only related to the parameters of code lengths and dependant to the element values in the parity matrix. Therefore, all scheduling are preprocessed offline without introducing much overhead in the online encoding.

Heuristic Matching. Similar ideas of reusing intermediate results are proposed in matching-based scheduling [7]. The idea is generalized from only reusing the result parity bits to reusing as many common parts in the intermediate results as possible. However, the proposed method focuses mainly on a pair of two data bits rather than three or more.

The main design approach is to represent the demand relation in encoding as a graph, where each vertex stands for an input data bit and each edge represents that some parity bits require an XOR operation between these two bits. After the graph is constructed, a maximum cardinality matching algorithm serves as a heuristic in greedily finding the most commonly-used data bit pairs (i.e., the edges in the graph). These pairs are calculated and the corresponding edges are removed from the graph. Then the scheduler recursively calls such matching on the remaining graph and repeat this process.

Similarly to smart scheduling [18], heuristic matching is also preprocessed offline. Meanwhile, two variants has been proposed for the matching (with an example in Figure 3):

(1) Unweighted Matching: All edges are viewed as possessing the same weights.
(2) Weighted Matching: The edge weights are set to be the opposite number of the sum of the degrees of both end nodes, in order to keep the dense nodes for the matching in the future rounds.

Architecture-aware. The final family of EC computation optimizations introduced in this work is highly dependant on architectures on modern computer systems. One typical approach [19] is to use single instruction multiple data (SIMD) in modern CPUs to directly vectorize finite field operations.

Another approach is to design cache-friendly schedules [14]. A data chunk is accessed only once sequentially, where all related parities are updated. This method tremendously increases efficiency by avoiding cache misses.

2.4 Discussions

Despite numerous developments in both SDC and EC communities, SDC during EC process is completely ignored in the literature.

SDC has only been examined under the context of storage, or generic computation, but never the erasure coding computation itself. This will increasingly become a concern, since EC is usually the last line of defense to protect data integrity.

On the EC front, algorithms and heuristics (like the ones introduced in Section 2.3) solely considered EC overhead (time, memory) and the amount of redundancy it provides. In the presence of SDC, these algorithms would present different error patterns, and thereby induces different detection/coorection methods and overhead. For example, both Smart Scheduling and Heuristic Matching explained in Section 2.3 utilize intermediate results to compute erasure codes faster. Depending on where among the intermediate results SDC happens, the error can propagate to multiple locations of the parity result.

This project will analyze these error scenarios in erasure coding techniques, specifically those scheduling based optimizations. We aim to learn the error patterns that are specific to different heuristics, and develop ways to detect/correct them.

3 PRELIMINARY

Before we head into our analysis, method design and experiments, we first introduce here the preliminaries and some basic building blocks in this section.

3.1 Model of SDC

As mentioned earlier, SDC can originate at different levels of the computation. In this work, we consider the type of corruptions that have sufficiently large mean time between failures(MTBF). We assume there could be at most one SDC (i.e. one bit flip) during one session of coding. We argue this assumption is reasonable since even for repeatable SDCs, it oftentimes requires exactly the same data on the same instruction to trigger (e.g. 1231343*122341 with AVX vectorization).

Additionally, we only consider SDC that happens at the CPU instruction level, including memcpy and arithmetic computations. We assume the storage and computation elsewhere are intact.

Lastly, our work focuses on SDC during the encoding process (rather than decoding). Encoding and decoding process are symmetric in their computation. We argue the results of our work would be largely relevant for SDC during the decoding process. The exact trade-off of performing SDC detection/correction at the encoding versus decoding is left for future work.

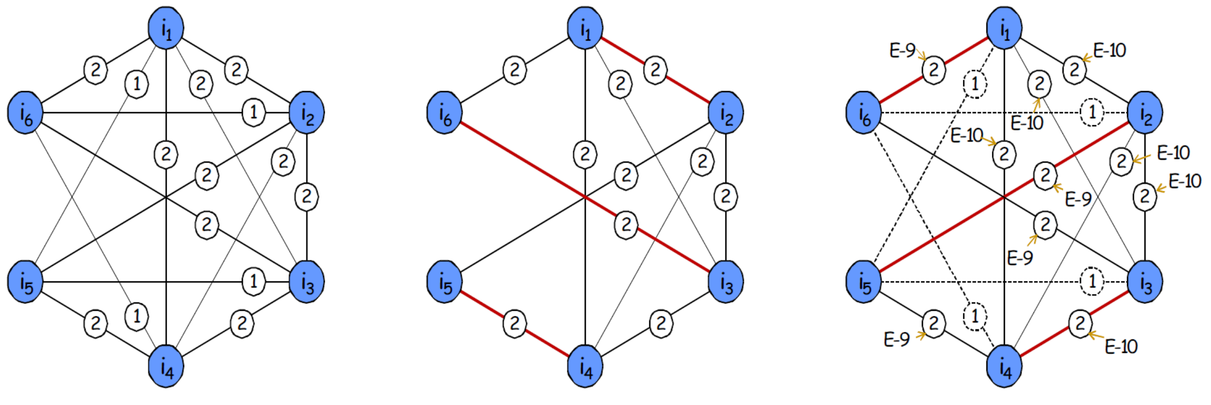

(a) graph with edge counters
(b) approach I: cardinality matching
(c) approach II: weighted matching

Figure 3: Example of heuristic matching from [7]. *E* here is an arbitrary significantly-large constant.

3.2 Model of EC

In this work, we chose three classes of common erasure coding methods to study the impact and detection of SDC – Matrix encoding using Maximum Distance Code, Bitmatrix encoding, and Scheduled bimatrix encoding. All three classes of algorithms had been implemented in the Jerasure library, on which we base our implementation.

3.3 Heavy Path Decomposition

Heavy path decomposition [22] (or *heavy-light decomposition*) is a method to decompose a rooted tree/forest into a set of paths. It is used as a basic building block in our detection method.

A *heavy edge* of a non-leaf node is defined as the edge connecting the node with its child who has the largest number of descendants in its subtree. All other edges connecting to the other children are called *light edges*.

In a heavy path decomposition, each non-leaf node chooses its heavy edge. These heavy edges will eventually form a set of paths, called *heavy paths*. Each non-leaf node belongs to exactly one heavy path. We can represent this path decomposition into a *heavy path tree*. Each heavy path / isolated leaf node is compressed into a compound node. The parent of this compound node in the heavy path tree is the compound the node which contains the parent of the head of the corresponding heavy path in the original tree. Figure 4 provides an example of a heavy path decomposition and its corresponding heavy path tree.

Lemma 3.1. *In a rooted tree with n nodes, a path from the root to any leaf passes through at most* $\lceil \log_2 n \rceil$ *light edges.*

Proof. On this path, passing through a light edge means that the subtree size of the child node connected with this light edge is smaller to at least one of its siblings. Thus, the remaining tree size is reduced at least by a half. Therefore, there will be at most $\lceil \log_2 n \rceil$ passes through light edges until the tree size is reduced to one, i.e., reaching a leaf. □

Theorem 3.2. *For a rooted tree with n nodes, its corresponding heavy path tree has at most a height of* $\lceil \log_2 n \rceil$.

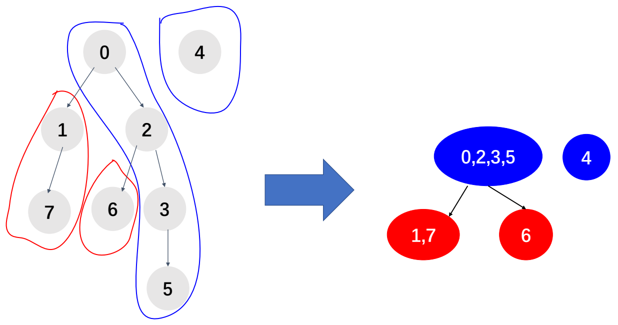


Figure 4: An example of heavy path decomposition on the left, and its corresponding heavy path tree (forest) on the right.

Proof. A path in the heavy path tree is equivalent to the set of light edges in the corresponding path from the root in the original tree to the corresponding leaf. Thus, the height can be bounded by Lemma 3.1. □

4 DESIGN FOR SDC DETECTION

4.1 SDC Error Propagation

In order to develop SDC detection methods, we first analyze error propagation patterns of SDC in the EC algorithms we consider. The following results are validated in experiments using our injection tools (described in Section 5).

Matrix Encoding. Matrix Encoding using Maximum Distance Code is done by multiplying Distribution matrix and Data matrix. Each element in the matrices is an element in $GF(2^w)$. One SDC (one bit flip) should result in exactly one bit error in the resulting code.

Bitmatrix Encoding. Bitmatrix encoding is almost identical as matrix encoding, except there is an extra step of encoding the data into bitmatrices. Each element in $GF(2^w)$ is represented by a $w \times w$ bitmatrix. The overall computation exhibits the same error

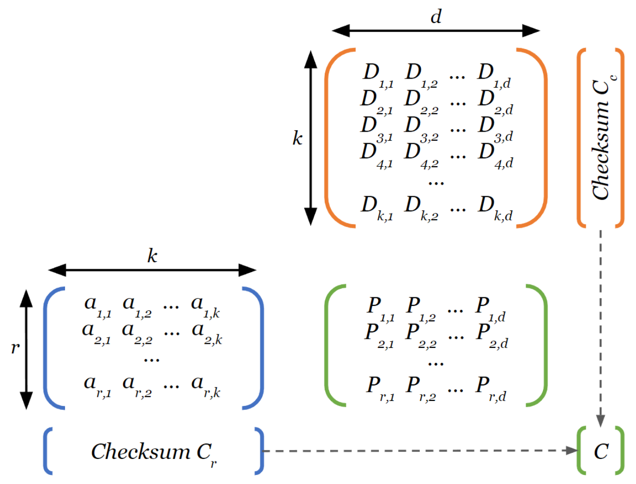


Figure 5: Starwman Detection Algorithm inspired by ABFT [9]

propagation pattern as matrix encoding – one SDC (one bit flip) should result in exactly one bit error in the resulting code.

Scheduled Bitmatrix Encoding. As explained earlier, scheduled bitmatrix encoding reuses intermediate results between the computation of multiple parity bits. One SDC (one bit flip) should result in various bit error in the resulting code. The exact number of erroneous bits in the final code depends on where SDC happened in the overall computation.

4.2 Strawman Detection

The most straightforward way to perform the detection is to redo the computation and compare the results. However, this comes with risk since some hardware deficiencies cause permanent bit-flips so the same computation would always lead to the same SDCs, making the errors impossible to detect. A better way is to perform the same computation on different hardware since two pieces of hardware manifesting the exact same SDC is extremely rare, but the algorithm still suffers from a 1× detection overhead and potential data transfer overhead that comes along.

Inspired by Algorithm-Based Fault Tolerance (ABFT) [9], we realize that erasure coding computation can use the same methods to detect matrix multiplication because finite-field arithmetic shares the same associative and commutative properties as normal addition and multiplication. As illustrated by Figure 5, the algorithm respectively calculates the row checksum of the encoding matrix (C_r), the column checksum of the data matrix (C_c), and the overall checksum of the parity matrix (C). Then the equation $C_r \cdot C_c = C$ should hold was there an even number of errors happening. Please refer to [9] for detailed proof of the method. Therefore, this method is very efficient in detecting a single bit-flip, which is mostly the case. Compared to the $O(r \cdot d \cdot k \cdot w)$ computation overhead of re-computation, a great overhead reduction is achieved by ABFT. The row and column checksum takes together $O((r + d) \cdot k \cdot w)$ to compute, the overall checksum $O(r \cdot d \cdot w)$. Therefore, the overall overhead is reduced to $O(((r \cdot k + d \cdot k + r \cdot d) \cdot w)$.

$C0 = D0 \oplus D2 \oplus D6$	$C4 = D0 \oplus D3$
$C1 = C0 \oplus D7$	$C5 = C3 \oplus D3 \oplus D4$
$C2 = C0 \oplus D3 \oplus D4$	$C6 = C2 \oplus D1$
$C3 = C2 \oplus D5$	$C7 = C1 \oplus D2$

Figure 6: An example of a smart schedule, whose data dependency graph is Figure 4. C_is are result data-bits to calculate; D_is are original data-bits.

4.3 Detection for Smart Scheduling

For methods that use offline scheduling to accelerate the encoding/decoding, partially computed results are reused in later computation of result data-bits. This means that a single root cause of silent data corruption might be propagated to several result data-bits, providing an opportunity to detect less target bits in order to spot an error.

We start with the design of SDC detection for smart scheduling [18]. Smart scheduling reuses the eventual results that will appear in the final parity matrix as the only source of the partially computed intermediate results. Meanwhile, it constrains that each result data-bit uses at most one previously computed data-bit to accelerate its computation (so that the search space for the offline scheduling is not oversized). With these constraints, smart scheduling uses an heuristic greedy algorithm for the offline scheduling by repeatedly picking up a parity row that uses the least number of XOR operation and refreshing the number of XORs of uncomputed rows each time a new row is picked/scheduled.

We can build a dependency graph for the result data-bits in the parity matrix based on the smart schedule. Each node in the dependency graph is a result data-bit and edges are the relationship of reusing partial results in the schedule. Figure 4 is an example of a dependency graph, whose corresponding schedule is Figure 6. As we can observe, the constraints made by smart scheduling ensures that the dependency graph is always a **tree**/**forest**.

The tree/forest structured schedule has good properties when detecting potential SDC. With the assumption that there exists at most one SDC per process, a key observation is that a root cause of a bit-flip in a node in the dependency tree will always be propagated to all its descendants. Reversely expressed, if we can validate that a node in the dependency tree is correctly computed, we can validate that all its ancestors are also correct.

This observation provides us with several principles when designing a detection method for SDCs. First, in smart scheduling, validating a leaf node in the dependency graph is enough to validate all nodes on the path from the root to this leaf. Second, for two nodes i and j, validating a checksum bit ($i \oplus j$) is enough to validate all i's ancestors and j's ancestors, except their common ancestors. Therefore, when we try to design detection for SDC on smart scheduling, one typical approach is to decompose the dependency tree/forest into paths towards the leaf nodes and only detect these leaf nodes, which is exactly what heavy path decomposition does.

With the previous analysis, we propose here our detection method on smart scheduling. First, we decompose the dependency graph into its heavy path forest (with largest height $h \leq \lceil \log_2 n \rceil$). Next, we calculate a total of h checksum bits $s_{1,\ldots,h}$, where each s_i XORs all the compound nodes on the i-th level in the heavy path forest. Take Figures 4 and 6 as an example, we only need 2 checksum bits in total, where $s_1 = C_4 \oplus C_5$ and $s_2 = C_7 \oplus C_6$. These two steps can both be carried out during the offline scheduling process, and we can offline produce the schedules of calculating $s_{1,\ldots,h}$.

For detection, we only need to validate that the $s_{1,\ldots,h}$ are all correct between the parity matrix and the calculated schedules from the original data matrix. The number of checksum bits is reduced to $O(\log r)$ from $O(r)$ in the strawman detection.

4.4 Detection for Generalized Scheduling

Typically, current scheduling-based methods still only reuses the eventual results that will appear in the final parity matrix as the only source of the partially computed intermediate results (such as in [7]). However, each result bit might use multiple previously calculated bits, instead of just one as in smart scheduling [18]. This results in the dependency graph being a **directed acyclic graph** (**DAG**) instead of a forest.

Designing a well-bounded decomposition for a DAG as in Section 4.3 is non-trivial. We save this for future work and only propose here a heuristic greedy method for the detection. The main idea is to greedily construct checksum bits to be detected, trying to add the node that enlarges the coverage of the current checksum bit to the largest extent. Please refer to the pseudocodes in Algorithm 1 for details.

Algorithm 1 Detection for scheduling-based methods

Input: Node array n [...]; Dependency graph d.
Output: Array res [...], each res $[i]$ a set structure including all rows in parity matrix to XOR in the i-th checksum bit.

```
1: Initiate array $diff$ [...]
2: Mark all $n$ [...] as uncovered
3: $op \leftarrow 0$
4: while $n$ [...] not all covered do
5:     Initiate empty set structure $set\_t$
6:     Mark all uncovered nodes as unchosen
7:     while Uncovered nodes not all chosen do ▹ This is loop $A$
8:         Set $diff$ [...] all to 0
9:         for all unchosen nodes $n$ $[i]$ do
10:            $diff$ $[i]$ ← (number of $n$ $[i]$'s ancestor not in $set\_t$)−(number of $n$ $[i]$'s ancestor in $set\_t$)
11:        $j \leftarrow augmax(diff$ [...]$)$
12:        if $diff$ $[j] > 0$ then
13:            for all $n$ $[j]$'s ancestor $nn$ do
14:                if $nn$ is in $set\_t$ then
15:                    $set\_t$.remove($nn$)
16:                else
17:                    $set\_t$.insert($nn$)
18:            Mark $n$ $[j]$ as chosen
19:        else
20:            Break loop $A$
21:    Mark all nodes in $set\_t$ as covered
22:    $res$ $[op]$.insert(all chosen nodes)
23:    $op \leftarrow op + 1$
24: return $res$ [...]
```

5 IMPLEMENTATION

We implemented our SDC injection tool, as well as SDC detection methods as an extension of a highly-optimized library for EC computation, Jerasure [20]. Jerasure makes use of GF-complete, an optimized library for Galois Field arithmetic. High-level implementation of the Jerasure library is the following:

Given size parameters for data and coding, matrix encoding routines generate the distribution matrix according to Reed-Solomon coding, or Cauchy Reed-Solomon coding (offline). During the encoding, data matrix is then multiplied to the distribution matrix using primitives in GF-complete (online). Along the way, the library also tabulates the numbers of bits that performed xor, memcpy, GF arithmetic, respectively.

Bitmatrix encoding routines take an extra step of encoding matrices into bitmatrices.

For scheduled bitmatrix encoding, instead of directly performing matrix multiplication, Jerasure first generates an optimized schedule of computing all the parity bits (offline). Then during encoding, the schedule is interpreted one by one to compute the corresponding erasure code (online).

The implementation is in C.

5.1 SDC injection

For SDC injection, we take advantage of the numbers of bits that performed xor, memcpy, GF arithmetic that are tabulated along the encoding routines. Given a set of size parameters, we first encode as usual to get a correct, baseline code. Then for each bit that performed xor, memcpy, GF arithmetic, we reinvoke the coding routines and inject the bitflip. This way, we inject SDC for each possible position of bitflips. After each iteration of SDC injection, we compare the SDC-injected erasure code against the baseline code.

5.2 Detection

For detection, the implementation also consists of of offline and online components. In the offline component, in addition to encoding schedule, an optimized detection schedule is generated according to the algorithm described in Section 4.3. In the online component, the detection schedule is interpreted one by one, similar to the encoding schedule. For each operation in the detection schedule, the corresponding checksum is computed and checked.

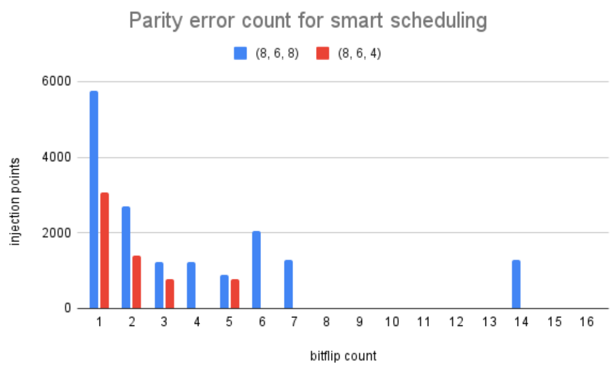


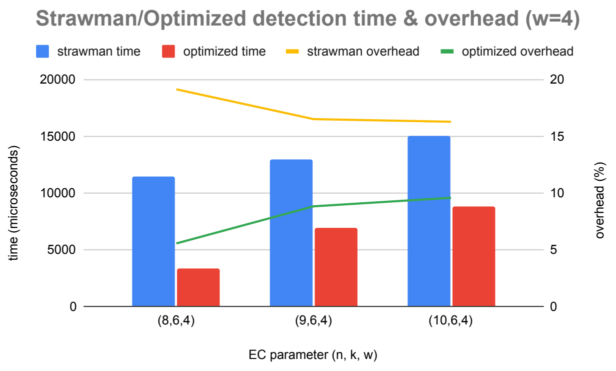
Figure 7: Injection point count against parity matrix bitflips for EC $(8, 6, 8)$ & $(8, 6, 4)$, Note that Jerasure truncates the identity matrix in the encoding matrix. Therefore for these configurations $(8, 6, 8)$ and $(8, 6, 4)$, the converted bitmatrices are of size $(2 \times 8, 6 \times 8)$ and $(2 \times 4, 6 \times 4)$. Since the smart schedule is performed on the granularity of rows, the resulting schedule tree has theoretical maximum heights of 16 and 8 respectively.

6 EVALUATION

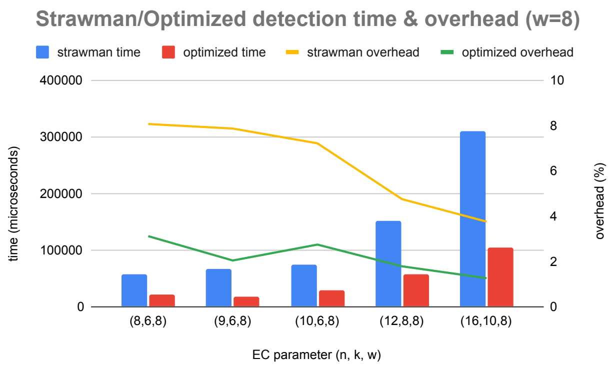
We evaluated the impact of silent data corruption on erasure coding through error injections in the various erasure coding implementation and optimization methods mentioned in Section 2.3. We performed the injections on a well-established erasure coding library, Jerasure [17]. Jerasure covers most of the optimization techniques we have discussed, namely naive encoding, bitmatrix encoding, and smart scheduling. We injected errors into all the computation routines. Since the computation scheme is static, we can perform a scan of all possible injection locations to verify the corresponding error propagation. For performance, we compared the overhead of the naive detection algorithm (performing duplicate computations of the entire encoding/decoding process), the strawman method, and optimized detection for smart scheduling to explore the tradeoff between detection completeness and computation overhead. We ran all our experiments on an eight-core Intel Xeon D-1548 at 2.0 GHz with 64 GB of memory.

6.1 Propagation of SDC

As mentioned earlier, the smart schedule is statically offline, so we can iterate over all possible points in the computation that a bit-flip can happen and manually inject errors over each location. The results of the common matrix and bitmatrix computation method are straightforward, each bit-flip is guaranteed to cause one error in the parity matrix. As shown in Figure 7, the result of smart scheduling is also as expected – a single injection can cause multiple errors in the parties due to the propagation of errors. We can see that for both of the configurations, the fewer injection points cause a larger number of flips in the parties. This also matches our analysis since a large number of flips in the parity means the flip happened early down the root of the schedule tree, whereas in most cases, the flip (injection) happens at a later phase or the leaf of the schedule tree, incurring fewer errors into the result parities.



(a) Runtime overhead with Galois field $GF(2^w)$ of $w = 4$



(b) Runtime overhead with Galois field $GF(2^w)$ of $w = 8$

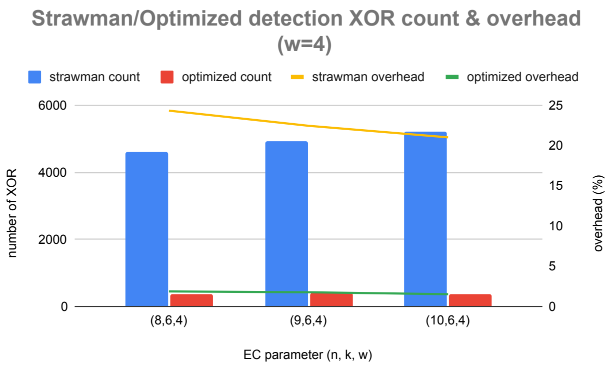
Figure 8: Runtime overhead of our detection methods. Bar chart is the absolute extra runtime used in detection, and the line chart is the overhead fraction compared with the encoding runtime.

6.2 Detection Methods

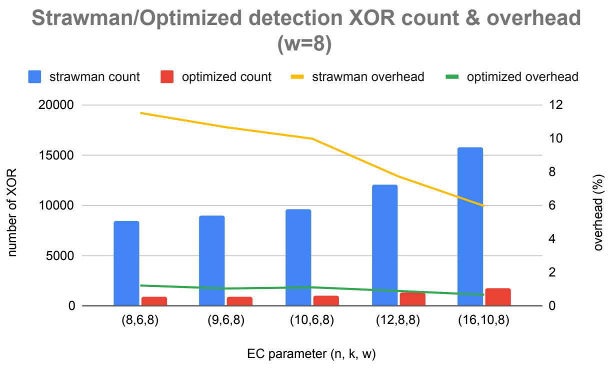
We test our detection methods over all injection cases in Section 6.1. Both of the methods can correctly detect all SDC existence in their targeted methods, not providing undetected errors or false positives.

We also evaluate the overhead of our two implemented detection methods. We test our detection methods on the same set of commonly-used EC parameters in practice as in the evaluation of [23]. The overheads are measured as the average fraction of extra runtime and XOR operations used for detection over all possible 1-bitflip SDC cases, respectively for the strawman and heavy-path decomposition based methods.

Runtime Overhead. Figure 8 illustrates the runtime of our detection methods, and their overhead compared with the encoding runtime without detections. Figure 8a are experiments carried out on commonly-used EC parameters tuples with Galois field $GF(2^w)$ of $w = 4$, while Figure 8b are the ones with $w = 8$. Overall, the strawman detection introduces an average overhead of 10.5%, with the worst case no more than 19.2%. Meanwhile, the detection for smart scheduling introduces an average overhead of 4.4%, with the worst case no more than 9.6%.



(a) **XOR operation overhead with Galois field $GF(2^w)$ of $w = 4$**



(b) **XOR operation overhead with Galois field $GF(2^w)$ of $w = 8$**

Figure 9: XOR operation overhead of our detection methods. Bar chart is the absolute number of extra XOR operations used in detection, and the line chart is the overhead fraction compared with the encoding operations.

XOR Operation Overhead. Figure 9 illustrates the number of XOR operations used in our detection methods, and their overhead compared with the encoding operations without detections. Figure 9a are experiments carried out on commonly-used EC parameters tuples with Galois field $GF(2^w)$ of $w = 4$, while Figure 9b are the ones with $w = 8$. Overall, the strawman detection introduces an average overhead of 14.2%, with the worst case no more than 24.4%. Meanwhile, the detection for smart scheduling introduces an average overhead of 1.25%, with the worst case no more than 1.86%.

Overall Results. With the same n and k, both encoding and detection takes more absolute time in larger ws. As an overall trend, the absolute runtime and XOR operations increases as n and k goes larger, while the overhead fraction of detections decreases. All of these phenomenons matches the theoretical analysis.

Discussions on Smart Scheduling Detection. There are several additional points we want to mention on smart scheduling detection. Although the strawman detection provides a quite monotonic trend of its overhead fraction when the bitmatrix size increases, the detection for smart scheduling gives out a Zigzag or even a minor increasing overhead in some of our experiment points. This is because the execution of smart scheduling detection is highly dependent on the shape of the dependency graph of the scheduling. Thus, if the scope of the axis of (n, k, w) tuples is not large enough, the local neighborhood might not guarantee a strictly decreasing overhead fraction.

We also notice that the runtime overhead of smart scheduling detection is not as low as the operation overhead. There exists at least two potential reasons. First, our evaluation of operation overhead only takes XOR operations into account, neglecting other operations such as *memcpy*. Meanwhile, our implementation is not highly optimized. The data structure we used to represent dependency and checksum scheduling might cause random memory access. Due to the limited time span of a course project, we do not have time to further fix this. We might save this for future work, and only target this current project as an initial work illustrating that encoding-aware detections do bring much less overhead than oblivious ones (even when they are not highly optimized in implementation).

7 CONCLUSION

Erasure coding has been a widely-used method to improve system robustness with unreliable storage. However, it has been increasingly threatened by silent data corruptions. This work initiates potential future study in the effect of silent data corruptions on erasure coding. We implement an error injector tool on commonly-used libraries as a powerful aiding tool in erasure coding research, and model the propagation patterns of silent data corruptions in several prevalent encoding acceleration methods. We also propose three detection methods of silent data corruption – strawman for oblivious applications, heavy-path decomposition based one for smart scheduling, and heuristic greedy search for generalized scheduling. We implement two of them and prove that they bring low overheads to detect all potential errors. We believe these detection can also add robustness to the CPU part of the system and detect faulty components. Future work includes works on more acceleration methods in other libraries, and develop well-bounded methods for generalized scheduling.

REFERENCES

[1] Baumann, R. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability 5*, 3 (2005), 305–316.
[2] Blomer, J. An xor-based erasure-resilient coding scheme. *Technical report at ICSI* (1995).
[3] Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., and Patterson, D. A. Raid: High-performance, reliable secondary storage. *ACM Comput. Surv. 26*, 2 (jun 1994), 145–185.
[4] Dixit, H. D., Pendharkar, S., Beadon, M., Mason, C., Chakravarthy, T., Muthiah, B., and Sankar, S. Silent data corruptions at scale. *arXiv preprint arXiv:2102.11245* (2021).
[5] Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *Proceedings of the nineteenth ACM symposium on Operating systems principles* (2003), pp. 29–43.
[6] Hochschild, P. H., Turner, P., Mogul, J. C., Govindaraju, R., Ranganathan, P., Culler, D. E., and Vahdat, A. Cores that don't count. In *Proceedings of the Workshop on Hot Topics in Operating Systems* (2021), pp. 9–16.
[7] Huang, C., Li, J., and Chen, M. On optimizing xor-based codes for fault-tolerant storage applications. In *2007 IEEE Information Theory Workshop* (2007), pp. 218–223.
[8] Huang, C., Simitci, H., Xu, Y., Ogus, A., Calder, B., Gopalan, P., Li, J., and Yekhanin, S. Erasure coding in windows azure storage. In *Presented as part of the 2012 USENIX Annual Technical Conference (USENIX ATC 12)* (2012), pp. 15–26.
[9] Huang, K.-H., and Abraham, J. A. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers C-33*, 6 (1984), 518–528.
[10] Iturbe, X., Venu, B., and Ozer, E. Soft error vulnerability assessment of the real-time safety-related arm cortex-r5 cpu. In *2016 IEEE International Symposium*

on Defect and Fault Tolerance in VLSI and Nanotechnology Systems (DFT) (2016), pp. 91–96.

[11] Kishani, M., Ahmadian, S., and Asadi, H. A modeling framework for reliability of erasure codes in ssd arrays. *IEEE Transactions on Computers 69*, 5 (2020), 649–665.

[12] Leray, J. Effects of atmospheric neutrons on devices, at sea level and in avionics embedded systems. *Microelectronics Reliability 47*, 9-11 (2007), 1827–1835.

[13] Luby, M., and Zuckermank, D. An xor-based erasure-resilient coding scheme. *Tech Report, Tech. Rep.* (1995).

[14] Luo, J., Shrestha, M., Xu, L., and Plank, J. S. Efficient encoding schedules for xor-based erasure codes. *IEEE Transactions on Computers 63*, 9 (2013), 2259–2272.

[15] MacWilliams, F. J., and Sloane, N. J. A. *The theory of error-correcting codes*, vol. 16. Elsevier, 1977.

[16] Plank, J., and Xu, L. Optimizing cauchy reed-solomon codes for fault-tolerant network storage applications. In *Fifth IEEE International Symposium on Network Computing and Applications (NCA'06)* (2006), pp. 173–180.

[17] Plank, J. S. Jerasure: A library in C/C++ facilitating erasure coding for storage applications. Tech. Rep. CS-07-603, University of Tennessee, September 2007.

[18] Plank, J. S. The raid-6 liberation codes. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies* (USA, 2008), FAST'08, USENIX Association.

[19] Plank, J. S., Greenan, K. M., and Miller, E. L. Screaming fast galois field arithmetic using intel simd instructions. In *FAST* (2013), pp. 299–306.

[20] Plank, J. S., Simmerman, S., and Schuman, C. D. Jerasure: A library in C/C++ facilitating erasure coding for storage applications - Version 1.2. Tech. Rep. CS-08-627, University of Tennessee, August 2008.

[21] Reed, I. S., and Solomon, G. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics 8*, 2 (1960), 300–304.

[22] Sleator, D. D., and Tarjan, R. E. A data structure for dynamic trees. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing* (New York, NY, USA, 1981), STOC '81, Association for Computing Machinery, p. 114–122.

[23] Zhou, T., and Tian, C. Fast erasure coding for data storage: A comprehensive study of the acceleration techniques. *ACM Transactions on Storage (TOS) 16*, 1 (2020), 1–24.