Optimal Resilient Sorting and Searching in the Presence of Memory Faults

Irene Finocchi[1], Fabrizio Grandoni[1], and Giuseppe F. Italiano[2]

[1] Dipartimento di Informatica, Università di Roma "La Sapienza", Via Salaria 113, 00198 Roma, Italy. `[finocchi,grandoni]@di.uniroma1.it`
[2] Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", Via del Politecnico 1, 00133 Roma, Italy. `italiano@disp.uniroma2.it`

Abstract. We investigate the problem of reliable computation in the presence of faults that may arbitrarily corrupt memory locations. In this framework, we consider the problems of sorting and searching in optimal time while tolerating the largest possible number of memory faults. In particular, we design an $O(n \log n)$ time sorting algorithm that can optimally tolerate up to $O(\sqrt{n \log n}\,)$ memory faults. In the special case of integer sorting, we present an algorithm with linear expected running time that can tolerate $O(\sqrt{n}\,)$ faults. We also present a randomized searching algorithm that can optimally tolerate up to $O(\log n)$ memory faults in $O(\log n)$ expected time, and an almost optimal deterministic searching algorithm that can tolerate $O((\log n)^{1-\epsilon})$ faults, for any small positive constant ϵ, in $O(\log n)$ worst-case time. All these results improve over previous bounds.

1 Introduction

The need of reliable computations in the presence of memory faults arises in many important applications. In fault-based cryptanalysis, for instance, some recent optical and electromagnetic perturbation attacks [4, 24] work by manipulating the non-volatile memories of cryptographic devices, so as to induce very timing-precise controlled faults on given individual bits: this forces the devices to output wrong ciphertexts that may allow the attacker to determine the secret keys used during the encryption.

Applications that make use of large memory capacities at low cost also incur into problems of memory faults and reliable computation. Indeed, the unpredictable failures known as *soft memory errors* tend to increase with memory size and speed [12, 19, 25]. Although the number of faults could be reduced by means of error checking and correction circuitry, this imposes non-negligible costs in terms of performance (as much as 33%), size (20% larger areas), and money (10% to 20% more expensive chips). For these reasons, this is not typically implemented in low-cost memories. Data replication is a natural approach to protect against destructive memory faults. However, it can be very inefficient in highly dynamic contexts or when the objects to be managed are large and complex: copying such objects can indeed be very costly, and in some cases we might not even know how to do this (for instance, when the data is accessed through pointers, which are moved around in memory instead of the data itself, and the algorithm relies on user-defined access functions). In these cases, we cannot assume either the existence of *ad hoc* functions for data replication or the definition of suitable encoding mechanisms to maintain a reasonable storage cost. As an example, consider Web search engines, which need to store and process huge data sets (of the order of Terabytes), including inverted indices which have to be maintained sorted for fast document access: for such large data structures, even a small failure probability can result in few bit flips in the index, that may become responsible of erroneous answers to keyword searches [9, 13]. In all these scenarios, it makes sense to assume that it must be the algorithms themselves, rather than specific hardware/software fault detection and correction mechanisms, in charge of dealing with memory faults. Informally, we have a *memory fault* when the correct value that should be stored in a memory location gets altered because of a failure, and we say that an algorithm is resilient to memory faults if, despite the corruption of some memory values before or during its execution, the algorithm is nevertheless able to get a correct output (at least) on the set of uncorrupted values.

The problem of computing with unreliable information has been investigated in a variety of different settings, including the *liar model* [1, 5, 10, 15, 16, 20–23, 26], fault-tolerant sorting networks [2, 17, 18, 27], resiliency of pointer-based data structures [3], parallel models of computation with faulty memories [7, 8, 14]. In [11], we introduced a *faulty-memory random access machine*, i.e., a random access machine whose memory locations may suffer from memory faults. In this model, an adversary may corrupt up to δ memory words throughout the execution of an algorithm. The algorithm cannot distinguish corrupted values from correct ones and can exploit only $O(1)$ *safe* memory words, whose content gets never corrupted. Furthermore, whenever it reads some memory location, the read operation will temporarily store its value in the safe memory. The adversary is adaptive, but has no access to information about future random choices of the algorithm: in particular, loading a random memory location in safe memory can be considered an atomic operation.

In this paper we address the problems of resilient sorting and searching in the faulty-memory random access machine. In the *resilient sorting* problem we are given a sequence of n keys that need to be sorted. The value of some keys can be arbitrarily corrupted (either increased or decreased) during the sorting process. The resilient sorting problem is to order correctly the set of uncorrupted keys. This is the best that we can achieve in the presence of memory faults, since we cannot prevent keys corrupted at the very end of the algorithm execution from occupying wrong positions in the output sequence. In the *resilient searching* problem we are given a sequence of n keys on which we wish to perform membership queries. The keys are stored in increasing order, but some keys may be corrupted (at any instant of time) and thus may occupy wrong positions in the sequence. Let x be the key to be searched for. The resilient searching problem is either to find a key equal to x, or to determine that there is no correct key equal to x. Also in this case, this is the best we can hope for, because memory faults can make x appear or disappear in the sequence at any time.

In [11] we contributed a first step in the study of resilient sorting and searching. In particular, we proved that any resilient $O(n \log n)$ comparison-based sorting algorithm can tolerate the corruption of at most $O(\sqrt{n \log n}\,)$ keys and we presented a resilient algorithm that tolerates $O(\sqrt[3]{n \log n}\,)$ memory faults. With respect to searching, we proved that any $O(\log n)$ time deterministic searching algorithm can tolerate at most $O(\log n)$ memory faults and we designed an

$O(\log n)$ time searching algorithm that can tolerate up to $O(\sqrt{\log n})$ memory faults.

The main contribution of this paper is to close the gaps between upper and lower bounds for resilient sorting and searching. In particular:

- We design a resilient sorting algorithm that takes $O(n \log n + \delta^2)$ worst-case time to run in the presence of δ memory faults. This yields an algorithm that can tolerate up to $O(\sqrt{n \log n})$ faults in $O(n \log n)$ time: as proved in [11], this bound is optimal.
- In the special case of integer sorting, we present a randomized algorithm with expected running time $O(n + \delta^2)$: thus, this algorithm is able to tolerate up to $O(\sqrt{n})$ memory faults in expected linear time.
- We prove an $\Omega(\log n + \delta)$ lower bound on the expected running time of resilient searching algorithms: this extends the lower bound for deterministic algorithms given in [11].
- We present an optimal $O(\log n + \delta)$ time randomized algorithm for resilient searching: thus, this algorithm can tolerate up to $O(\log n)$ memory faults in $O(\log n)$ expected time.
- We design an almost optimal $O(\log n + \delta^{1+\epsilon'})$ time deterministic searching algorithm, for any constant $\epsilon' \in (0, 1]$: this improves over the $O(\log n + \delta^2)$ bound of [11] and yields an algorithm that can tolerate up to $O((\log n)^{1-\epsilon})$ faults, for any small positive constant ϵ.

Notation. We recall that δ is an upper bound on the total number of memory faults. We also denote by α the *actual* number of faults that happen during a specific execution of an algorithm. Note that $\alpha \leq \delta$. We say that a key is *faithful* if its value is never corrupted by any memory fault, and *faulty* otherwise. A sequence is *faithfully ordered* if its faithful keys are sorted, and *k-unordered* if there exist k (faithful) keys whose removal makes the remaining subsequence faithfully ordered. Given a sequence X of length n, we use $X[a\,;\,b]$, with $1 \leq a \leq b \leq n$, as a shortcut for the subsequence $\{X[a], X[a+1], \ldots, X[b]\}$. Two keys $X[p]$ and $X[q]$, with $p < q$, form an *inversion* in the sequence X if $X[p] > X[q]$: note that, for any two keys forming an inversion in a faithfully ordered sequence, at least one of them must be faulty. A sorting or merging algorithm is called *resilient* if it produces a faithfully ordered sequence.

2 Optimal Resilient Sorting in the Comparison Model

In this section we describe a resilient sorting algorithm that takes $O(n \log n + \delta^2)$ worst-case time to run in the presence of δ memory faults. This yields an $O(n \log n)$ time algorithm that can tolerate up to $O(\sqrt{n \log n})$ faults: as proved in [11], this bound is optimal if we wish to sort in $O(n \log n)$ time, and improves over the best known resilient algorithm, which was able to tolerate only $O(\sqrt[3]{n \log n})$ memory faults [11]. We first present a fast resilient merging algorithm, that may nevertheless fail to insert all the input values in the faithfully ordered output sequence. We next show how to use this algorithm to solve the resilient sorting problem within the claimed $O(n \log n + \delta^2)$ time bound.

The Purifying Merge Algorithm. Let X and Y be the two faithfully ordered sequences of length n to be merged. The merging algorithm that we are going to describe produces a faithfully ordered sequence Z and a disordered fail sequence

F in $O(n+\alpha\,\delta)$ worst-case time. It will be guaranteed that $|F| = O(\alpha)$, i.e., that only $O(\alpha)$ keys can fail to get inserted into Z.

The algorithm, called `PurifyingMerge`, uses two auxiliary input buffers of size $(2\delta+1)$ each, named $\mathcal{X}$ and $\mathcal{Y}$, and an auxiliary output buffer of size δ, named $\mathcal{Z}$. The input buffers $\mathcal{X}$ and $\mathcal{Y}$ are initially filled with the first $(2\delta+1)$ values in X and Y, respectively. The merging process is divided into rounds: the algorithm maintains the invariant that, at the beginning of each round, both input buffers are full while the output buffer is empty (we omit here the description of the boundary cases). Each round consists of merging the contents of the input buffers until either the output buffer becomes full or an inconsistency in the input keys is found. In the latter case, we perform a *purifying step*, where two keys are moved to the fail sequence F. We now describe the generic round in more detail.

The algorithm fills buffer $\mathcal{Z}$ by scanning the input buffers $\mathcal{X}$ and $\mathcal{Y}$ sequentially. Let i and j be the running indices on $\mathcal{X}$ and $\mathcal{Y}$: we call $\mathcal{X}[i]$ and $\mathcal{Y}[j]$ the *top keys* of $\mathcal{X}$ and $\mathcal{Y}$, respectively. The running indices i and j, the top keys of $\mathcal{X}$ and $\mathcal{Y}$, and the last key copied to $\mathcal{Z}$ are all stored in $O(1)$ size safe memory. At each step, we compare $\mathcal{X}[i]$ and $\mathcal{Y}[j]$: without loss of generality, assume that $\mathcal{X}[i] \leq \mathcal{Y}[j]$ (the other case being symmetric). We next perform an *inversion check* as follows: if $\mathcal{X}[i] \leq \mathcal{X}[i+1]$, $\mathcal{X}[i]$ is copied to $\mathcal{Z}$ and index i is advanced by 1 (note that the key copied to $\mathcal{Z}$ is left in $\mathcal{X}$ as well). If the inversion check fails, i.e., $\mathcal{X}[i] > \mathcal{X}[i+1]$, we perform a purifying step on $X[i]$ and $X[i+1]$: we move these two keys to the fail sequence F, we append two new keys from X at the end of buffer $\mathcal{X}$, and we restart the merging process of the buffers $\mathcal{X}$ and $\mathcal{Y}$ from scratch by simply resetting all the buffer indices (note that this makes the output buffer $\mathcal{Z}$ empty). Thanks to the comparisons between the top keys and to the inversion checks, the last key appended to $\mathcal{Z}$ is always smaller than or equal to the top keys of $\mathcal{X}$ and $\mathcal{Y}$ (considering their values stored in safe memory): we call this *top invariant*. When $\mathcal{Z}$ becomes full, we check whether all the remaining keys in $\mathcal{X}$ and $\mathcal{Y}$ (i.e., the keys not copied into $\mathcal{Z}$) are larger than or equal to the last key $\mathcal{Z}[\delta]$ copied into $\mathcal{Z}$ (*safety check*). If the safety check fails on $\mathcal{X}$, the top invariant guarantees that there is an inversion between the current top key $\mathcal{X}[i]$ of $\mathcal{X}$ and another key remaining in $\mathcal{X}$: in that case, we execute a purifying step on those two keys. We do the same if the safety check fails on $\mathcal{Y}$. If all the checks succeed, the content of $\mathcal{Z}$ is flushed to the output sequence Z and the input buffers $\mathcal{X}$ and $\mathcal{Y}$ are refilled with an appropriate number of new keys taken from X and Y, respectively.

Lemma 1. *The output sequence Z is faithfully ordered.*

Proof. We say that a round is *successful* if it terminates by flushing the output buffer into Z, and *failing* if it terminates by adding keys to the fail sequence F. Since failing rounds do not modify Z, it is sufficient to consider successful rounds only. Let $\mathcal{X}'$ and X' be the remaining keys in $\mathcal{X}$ and X, respectively, at the end of a successful round. The definition of $\mathcal{Y}'$ and Y' is similar. We will show that: (1) buffer $\mathcal{Z}$ is faithfully ordered; and (2) all the faithful keys in $\mathcal{Z}$ are smaller than or equal to the faithful keys in $\mathcal{X}' \cup X'$ and $\mathcal{Y}' \cup Y'$. The lemma will then follow by induction on the number of successful rounds.

(1) We denote by $\widetilde{\mathcal{Z}}[h]$ the value of the h-th key inserted into $\mathcal{Z}$ at the time of its insertion. The sequence $\widetilde{\mathcal{Z}}$ must be sorted, since otherwise an inversion check

would have failed at some point. It follows that $\mathcal{Z}$ is faithfully ordered, since $\widetilde{\mathcal{Z}}[h] = \mathcal{Z}[h]$ for each faithful key $\mathcal{Z}[h]$.

(2) Consider now the largest faithful key $z = \widetilde{\mathcal{Z}}[k]$ in $\mathcal{Z}$ and the smallest faithful key x in $\mathcal{X}' \cup X'$. We will show that $z \leq x$ (if one of the two keys does not exist, there is nothing to prove). Note that x must belong to $\mathcal{X}'$. In fact, all the faithful keys in $\mathcal{X}'$ are smaller than or equal to the faithful keys in X'. Moreover, either $\mathcal{X}'$ contains at least $(\delta + 1)$ keys (and thus at least one faithful key), or X' is empty. All the keys in $\mathcal{X}'$ are compared with $\widetilde{\mathcal{Z}}[\delta]$ during the safety check. In particular, $x \geq \widetilde{\mathcal{Z}}[\delta]$ since the safety check was successful. From the order of $\widetilde{\mathcal{Z}}$, we obtain $\widetilde{\mathcal{Z}}[\delta] \geq \widetilde{\mathcal{Z}}[k] = z$, thus implying $x \geq z$. A symmetric argument shows that z is smaller than or equal to the smallest faithful key y in $\mathcal{Y}' \cup Y'$.

We now summarize the performance of algorithm `PurifyingMerge`.

Lemma 2. *Algorithm* `PurifyingMerge`*, given two faithfully ordered sequences of length n, merges the sequences in $O(n + \alpha\,\delta)$ worst-case time. The algorithm returns a faithfully ordered sequence Z and a fail sequence F such that $|F| = O(\alpha)$.*

Proof. The faithful order of Z follows from Lemma 1. The two values discarded in each failing round form an inversion in one of the input sequences, which are faithfully ordered. Thus, at least one of such discarded values must be corrupted, proving that the number of corrupted values in F at any time is at least $|F|/2$. This implies that $|F|/2 \leq \alpha$ and that the number of failing rounds is bounded above by α. Note that at each round we spend time $\Theta(\delta)$. When the round is successful, this time can be amortized against the time spent to flush δ values to the output sequence. We therefore obtain a total running time of $O(n + \alpha\,\delta)$.

The Sorting Algorithm. We first notice that a naive resilient sorting algorithm can be easily obtained from a bottom-up iterative implementation of mergesort by taking the minimum among $(\delta + 1)$ keys per sequence at each merge step. We call this `NaiveSort`.

Lemma 3. *Algorithm* `NaiveSort` *faithfully sorts n keys in $O(\delta\, n \log n)$ worst-case time. The running time becomes $O(\delta\, n)$ when $\delta = \Omega(n^{\epsilon})$, for some $\epsilon > 0$.*

In order to obtain a more efficient sorting algorithm, we will use the following merging subroutine, called `ResilientMerge` (see also Figure 1). We first merge the input sequences using algorithm `PurifyingMerge`: this produces a faithfully ordered sequence Z and a disordered fail sequence F. We sort F with algorithm `NaiveSort` and produce a faithfully ordered sequence F'. We finally merge Z and F' using the algorithm `UnbalancedMerge` of [11], which has the following time bound:

Lemma 4. [11] *Two faithfully ordered sequences of length n_1 and n_2, with $n_2 \leq n_1$, can be merged faithfully in $O(n_1 + (n_2 + \alpha)\,\delta)$ worst-case time.*

We now analyze the running time of algorithm `ResilientMerge`.

Lemma 5. *Algorithm* `ResilientMerge`*, given two faithfully ordered sequences of length n, failthfully merges the sequences in $O(n + \alpha\,\delta)$ worst-case time.*

Proof. By Lemma 2, algorithm `PurifyingMerge` requires time $O(n + \alpha\,\delta)$ and produces a disordered fail sequence F of length $O(\alpha)$. Since $\delta = \Omega(\alpha)$, the total time required by algorithm `NaiveSort` to produce a faithfully sorted sequence F' from F is $O(\alpha\,\delta)$ by Lemma 3. Finally, by Lemma 4 algorithm `UnbalancedMerge` takes time $O(|Z|+(|F'|+\alpha)\,\delta) = O(n+\alpha\,\delta)$ to merge Z and F'. The total running time immediately follows.

Algorithm `ResilientMerge` has the property that only the keys corrupted while merging may be out of order in the output sequence. Hence, if we plug this algorithm into an iterative bottom-up implementation of mergesort, we obtain the following:

Theorem 1. *There is a resilient algorithm that sorts n keys in $O(n \log n + \alpha\,\delta)$ worst-case time and linear space.*

This yields an $O(n \log n)$ time resilient sorting algorithm that can tolerate up to $O(\sqrt{n \log n}\,)$ memory faults. As shown in [11], no better bound is possible.

3 Resilient Integer Sorting

In this section we consider the problem of faithfully sorting a sequence of n integers in the range $[0, n^c - 1]$, for some constant $c \geq 0$. We will present a randomized algorithm with expected running time $O(n+\delta^2)$: thus, this algorithm is able to tolerate up to $O(\sqrt{n}\,)$ memory faults in expected linear time. Our algorithm is a resilient implementation of (least significant digit) `RadixSort`, which works as follows. Assume that the integers are represented in base b, with $b \geq 2$. At the i-th step, for $1 \leq i \leq \lceil c \log_b n \rceil$, we sort the integers according to their i-th least significant digit using a linear time, stable bucket sorting algorithm (with b buckets). We can easily implement radix sort in faulty memory whenever the base b is constant: we keep an array of size n for each bucket and store the address of those arrays and their current length (i.e., the current number of items in each bucket) in the $O(1)$-size safe memory. Since there is only a constant number of buckets, we can conclude:

Lemma 6. *We can sort n polynomially bounded integers in $O(n \log n)$ worst-case time and linear space, while tolerating an arbitrary number of memory faults.*

Proof. The value v_i of the i-th digit of a given integer v influences the position of the integer itself only in the i-th step, when we have to determine to which bucket v must be appended. Let us call the value of v_i at that time its *virtual value.* Clearly, the algorithm correctly sorts the sequence according to the virtual values of the digits. The claim follows by observing that the real and virtual values of faithful elements are equal.

Unfortunately, in order to make radix sort run in linear time, we need $b = \Omega(n^\epsilon)$, for some constant $\epsilon \in (0, 1]$. However, if the number of buckets is not constant, we might need more than linear space. More importantly, $O(1)$ safe memory words would not be sufficient to store the initial address and the current length of the b arrays. We will now show how to overcome both problems. We store the b arrays contiguously, so that their initial addresses can be derived from a unique address β (which is stored in safe memory). However, we cannot store in the $O(1)$ safe memory the current length of each array. Hence, in the i-th step of radix sort, with $1 \leq i \leq \lceil c \log_b n \rceil$, we have to solve b instances of the following *bucket-filling* problem. We receive in an online fashion a sequence of $n' \leq n$ integers (faithfully) sorted up to the i-th least significant digit. We have to copy this input sequence into an array $\mathcal{B}_0$ whose current length cannot be stored in safe memory: $\mathcal{B}_0$ must maintain the same faithful order as the order in the input sequence.

In the rest of this section we will show how to solve the bucket-filling problem in $O(n' + \alpha\,\delta)$ expected time and $O(n' + \delta)$ space, where α is the actual number of memory faults occurring throughout the execution of the bucket-filling algorithm. This will imply the following theorem.

Theorem 2. *There is a randomized algorithm that faithfully sorts n polynomially bounded integers in $O(n + \alpha\,\delta)$ expected time. The space required is linear when $\delta = O(n^{1-\epsilon})$, for any small positive constant ϵ.*

Proof. It is sufficient to implement radix sort with base $b = \Theta(n^{\epsilon})$, via the randomized bucket-filling algorithm mentioned above. Consider the i-th step of radix sort. Let n_j denote the number of integers copied into bucket j, and let α_j be the actual number of memory faults affecting the execution of the j-th instance of the bucket-filling procedure. The running time of the i-th step is $O(\sum_{j=0}^{b-1}(n_j + \alpha_j\,\delta)) = O(n + \alpha\,\delta)$. The claim on the running time follows by observing that the total number of such steps is $O(\log_b n^c) = O(\log_{n^\epsilon} n) = O(1)$. The space usage is $O(\sum_{j=0}^{b-1}(n_j+\delta)) = O(n+b\,\delta)$. This is $O(n)$ when $\delta = O(n^{1-\epsilon})$.

The Bucket-Filling Problem. We first describe a deterministic bucket-filling algorithm with running time $O(n' + \alpha\,\delta^{1.5})$. The algorithm exploits the use of buffering techniques. We remark that the input integers are (faithfully) sorted up the i-th least significant digit and that we cannot store the current length of the buffers in safe memory. In order to circumvent this problem, we will use redundant variables, defined as follows. A *redundant $|p|$-index p* is a set of $|p|$ positive integers. The *value* of p is the majority value in the set (or an arbitrary value if no majority value exists). Assigning a value x to p means assigning x to all its elements: note that both reading and updating p can be done in linear time and constant space (using, e.g., the algorithm in [6]). If $|p| \geq 2\delta + 1$, we say that p is *reliable* (i.e., we can consider its value faithful even if p is stored in faulty memory). A *redundant $|p|$-pointer p* is defined analogously, with positive integers replaced by pointers.

Besides using redundant variables, we periodically restore the ordering inside the buffers by means of a (bidirectional) `BubbleSort`, which works as follows: we compare adjacent pairs of keys, swapping them if necessary, and alternately pass through the sequence from the beginning to the end and from the end to the beginning, until no more swaps are performed. Interestingly enough, `BubbleSort` is resilient to memory faults and its running time depends only on the disorder of the input sequence and on the actual number of faults occurring during its execution.

Lemma 7. *Given a k-unordered sequence of length n, algorithm* `BubbleSort` *faithfully sorts the sequence in $O(n + (k + \alpha)\,n)$ worst-case time.*

We now give a more detailed description of our bucket-filling algorithm. Besides the output array $\mathcal{B}_0$, we use two buffers to store temporarily the input keys: a buffer $\mathcal{B}_1$ of size $|\mathcal{B}_1| = 2\delta + 1$, and a buffer $\mathcal{B}_2$ of size $|\mathcal{B}_2| = 2\sqrt{\delta} + 1$. All the entries of both buffers are initially set to a value, say $+\infty$, that is not contained in the input sequence. We associate a redundant index p_i to each $\mathcal{B}_i$, where $|p_0| = |\mathcal{B}_1| = 2\delta + 1$, $|p_1| = |\mathcal{B}_2| = 2\sqrt{\delta} + 1$, and $|p_2| = 1$. Note that only p_0 is reliable, while p_1 and p_2 could assume faulty values. Both buffers and indexes are stored in such a way that their address can be derived from the unique address β stored in safe memory.

The algorithm works as follows. Each time a new input key is received, it is appended to $\mathcal{B}_2$. Whenever $\mathcal{B}_2$ is full (according to index p_2), we *flush* it as follows: (1) we remove any $+\infty$ from $\mathcal{B}_2$ and sort $\mathcal{B}_2$ with `BubbleSort` considering the i least significant digits only; (2) we append $\mathcal{B}_2$ to $\mathcal{B}_1$, and we update p_1 accordingly; (3) we reset $\mathcal{B}_2$ and p_2. Whenever $\mathcal{B}_1$ is full, we *flush* it in a similar way, moving its keys to $\mathcal{B}_0$. We flush buffer $\mathcal{B}_j$, $j \in \{1, 2\}$, also whenever we realize that the index p_j points to an entry outside $\mathcal{B}_j$ or to an entry of value different from $+\infty$ (which indicates that a fault happened either in p_j or in $\mathcal{B}_j$ after the last time $\mathcal{B}_j$ was flushed).

Lemma 8. *The algorithm above solves the bucket-filling problem in $O(n' + \alpha\,\delta^{1.5})$ worst-case time.*

Proof. To show the correctness, we notice that all the faithful keys eventually appear in $\mathcal{B}_0$. All the faithful keys in $\mathcal{B}_j$, $j \in \{1, 2\}$, at a given time precede the faithful keys not yet copied into $\mathcal{B}_j$. Moreover we sort $\mathcal{B}_j$ before flushing it. This guarantees that the faithful keys are moved from $\mathcal{B}_j$ to $\mathcal{B}_{j-1}$ in a first-in-first-out fashion.

Consider the cost paid by the algorithm between two consecutive flushes of $\mathcal{B}_1$. Let α' and α'' be the number of faults in $\mathcal{B}_1$ and p_1, respectively, during the phase considered. If no fault happens in either $\mathcal{B}_1$ or p_1 ($\alpha' + \alpha'' = 0$), flushing buffer $\mathcal{B}_1$ costs $O(|\mathcal{B}_1|) = O(\delta)$. If the value of p_1 is faithful ($\alpha'' \le \sqrt{\delta}$), the sequence is $O(\alpha')$-unordered: in fact, removing the corrupted values from $\mathcal{B}_1$ produces a sorted subsequence. Thus sorting $\mathcal{B}_1$ costs $O((1 + \alpha')\delta)$. Otherwise ($\alpha'' > \sqrt{\delta}$), the sequence $\mathcal{B}_1$ can be $O(\delta)$-unordered and sorting it requires $O((1 + \delta + \alpha')\delta) = O(\delta^2)$ time. Thus, the total cost of flushing buffer $\mathcal{B}_1$ is $O(n' + \alpha/\sqrt{\delta}\,\delta^2 + \alpha\,\delta) = O(n' + \alpha\,\delta^{1.5})$. Using a similar argument, we can show that the total cost of flushing buffer $\mathcal{B}_2$ is $O(n' + \alpha\,\delta)$. The claimed running time immediately follows.

The deterministic running time can be improved by choosing more carefully the buffer size and by increasing the number of buffers: specifically, with $\ell \ge 2$ buffers, we can achieve a $O(\ell\, n' + \alpha\,\delta^{2^\ell/(2^\ell - 1)})$ running time. The details of the multi-buffer algorithm will be included in the full paper. This yields an integer sorting algorithm with $O(n + \alpha\,\delta^{1+\epsilon})$ worst-case running time, for any small positive constant ϵ.

A Randomized Approach. We now show how to reduce the (expected) running time of the bucket-filling algorithm to $O(n' + \alpha\,\delta)$, by means of randomization. As we already observed in the proof of Lemma 8, a few corruptions in p_1 can lead to a highly disordered sequence $\mathcal{B}_1$. Consider for instance the following situation: we corrupt p_1 twice, in order to force the algorithm to write first δ faithful keys in the second half of B_1, and then other $(\delta+1)$ faithful keys in the first half of B_1. In this way, with $2(\sqrt{\delta}+1)$ corruptions only, one obtains an $O(\delta)$-unordered sequence, whose sorting requires $O(\delta^2)$ time. This can happen $O(\alpha/\sqrt{\delta})$ times, thus leading to the $O(\alpha\,\delta^{1.5})$ term in the running time.

The idea behind the randomized algorithm is to try to avoid such kind of pathological situations. Specifically, we would like to detect early the fact that many values after the last inserted key are different from $+\infty$. In order to do that, whenever we move a key from $\mathcal{B}_2$ to $\mathcal{B}_1$, we select an entry uniformly at random in the portion of $\mathcal{B}_1$ after the last inserted key: if the value of this entry is not $+\infty$, the algorithm flushes $\mathcal{B}_1$ immediately.

Lemma 9. *The randomized algorithm above solves the bucket-filling problem in* $O(n'+\alpha\,\delta)$ *expected time.*

Proof. Let α' and α'' be the number of faults in $\mathcal{B}_1$ and p_1, respectively, between two consecutive flushes of buffer $\mathcal{B}_1$. Following the proof of Lemma 8 and the discussion above, it is sufficient to show that, when we sort $\mathcal{B}_1$, the sequence to be sorted is $O(\alpha'+\alpha'')$-unordered in expectation. In order to show that, we will describe a procedure which obtains a sorted subsequence from $\mathcal{B}_1$ by removing an expected number of $O(\alpha'+\alpha'')$ keys.

First remove the α' corrupted values in $\mathcal{B}_1$. Now consider what happens either between two consecutive corruptions of p_1 or between a corruption and a reset of p_1. Let $\widetilde{p}_1$ be the value of p_1 at the beginning of the phase considered. By A and B we denote the subset of entries of value different from $+\infty$ after $\mathcal{B}_1[\widetilde{p}_1]$ and the subset of keys added to $\mathcal{B}_1$ in the phase considered, respectively. Note that, when A is large, the expected cardinality of B is small (since it is more likely to select randomly an entry in A). More precisely, the probability of selecting at random an entry of A is at least $|A|/|\mathcal{B}_1|$. Thus the expected cardinality of B is at most $|\mathcal{B}_1|/|A| = O(\delta/|A|)$.

The idea behind the proof is to remove A from $\mathcal{B}_1$ if $|A| < \sqrt{\delta}$, and to remove B otherwise. In both cases the expected number of keys removed is $O(\sqrt{\delta}\,)$. At the end of the process, we obtain a sorted subsequence of $\mathcal{B}_1$. Since p_1 can be corrupted at most $O(\alpha''/\sqrt{\delta}\,)$ times, the total expected number of keys removed is $O(\alpha'+\sqrt{\delta}\,\alpha''/\sqrt{\delta}\,) = O(\alpha'+\alpha'')$.

Saving Space. The bucket-filling algorithm described above uses $O(n+\delta)$ space, since each bucket is implemented via an array of size n. The space usage can be easily reduced to $O(n'+\delta)$ via doubling, without increasing the asymptotic running time. We initially impose $|\mathcal{B}_0| = \delta$. We store a $(2\delta+1)$-pointer p to $\mathcal{B}_0$ in faulty memory, such that its address can be derived from the unique address β stored in safe memory. When $\mathcal{B}_0$ is full, we create a new array $\mathcal{B}'_0$ of size $|\mathcal{B}'_0| = 2|\mathcal{B}_0|$, we copy $\mathcal{B}_0$ into the first $|\mathcal{B}_0|$ entries of $\mathcal{B}'_0$, and we make p point to $\mathcal{B}'_0$. The total cost of these operations is $O(n'+\delta)$, as well as the space complexity of the new bucket-filling algorithm.

4 Resilient Searching Algorithms

In this section we prove upper and lower bounds on the resilient searching problem. Namely, we first prove an $\Omega(\log n+\delta)$ lower bound on the expected running time, and then we present an optimal $O(\log n + \delta)$ expected time randomized algorithm. Finally, we sketch an $O(\log n + \delta^{1+\epsilon'})$ time deterministic algorithm, for any constant $\epsilon' \in (0, 1]$. Both our algorithms improve over the $O(\log n + \delta^2)$ deterministic bound of [11].

A Lower Bound for Randomized Searching. We now show that every searching algorithm, even randomized ones, which tolerates up to δ memory faults must have expected running time $\Omega(\log n + \delta)$ on sequences of length n, with $n \geq \delta$.

Theorem 3. *Every (randomized) resilient searching algorithm must have expected running time $\Omega(\log n + \delta)$.*

Proof. An $\Omega(\log n)$ lower bound holds even when the entire memory is safe. Thus, it is sufficient to prove that every resilient searching algorithm takes expected time $\Omega(\delta)$ when $\log n = o(\delta)$. Let $\mathcal{A}$ be a resilient searching algorithm. Consider the following (feasible) input sequence I: for an arbitrary value x, the first $(\delta + 1)$ values of the sequence are equal to x and the others are equal to $+\infty$. Let us assume that the adversary arbitrarily corrupts δ of the first $(\delta + 1)$ keys before the beginning of the algorithm. Since a faithful key x is left, $\mathcal{A}$ must be able to find it.

Observe that, after the initial corruption, the first $(\delta + 1)$ elements of I form an arbitrary (unordered) sequence. Suppose by contradiction that $\mathcal{A}$ takes $o(\delta)$ expected time. Then we can easily derive from $\mathcal{A}$ an algorithm to find a given element in an unordered sequence of length $\Theta(\delta)$ in sub-linear expected time, which is not possible (even in a safe-memory system).

Optimal Randomized Searching. In this section we present a resilient searching algorithm with optimal $O(\log n+\delta)$ expected running time. Let I be the sorted input sequence and x be the key to be searched for. At each step, the algorithm considers a subsequence $I[\ell; r]$. Initially $I[\ell; r] = I[1; n] = I$. Let $C > 1$ and $0 < c < 1$ be two constants such that $c\,C > 1$. The algorithm has a different behavior depending on the length of the current interval $I[\ell; r]$. If $r-\ell > C\delta$, the algorithm chooses an element $I[h]$ uniformly at random in the central subsequence of $I[\ell; r]$ of length $(r-\ell)c$, i.e., in $I[\ell'; r'] = I[\ell+(r-\ell)(1-c)/2;\ \ell+(r-\ell)(1+c)/2]$ (for the sake of simplicity, we neglect ceilings and floors). If $I[h] = x$, the algorithm simply returns the index h. Otherwise, it continues searching for x either in $I[\ell; h-1]$ or in $I[h+1; r]$, according to the outcome of the comparison between x and $I[h]$.

Consider now the case $r - \ell \leq C\delta$. Let us assume that there are at least 2δ values to the left of ℓ and 2δ values to the right of r (otherwise, it is sufficient to assume that $X[i] = -\infty$ for $i < 1$ and $X[i] = +\infty$ for $i > n$). If x is contained in $I[\ell - 2\delta; r + 2\delta]$, the algorithm returns the corresponding index. Else, if both the majority of the elements in $I[\ell - 2\delta; \ell]$ are smaller than x and the majority of the elements in $I[r; r + 2\delta]$ are larger than x, the algorithm returns no. Otherwise, at least one of the randomly selected values $I[h_k]$ must be faulty: in that case the algorithm simply restarts from the beginning.

Note that the variables needed by the algorithm require total constant space, and thus they can be stored in safe memory.

Theorem 4. *The algorithm above performs resilient searching in $O(\log n + \delta)$ expected time.*

Proof. Consider first the correctness of the algorithm. We will later show that the algorithm halts with probability one. If the algorithm returns an index, the answer is trivially correct. Otherwise, let $I[\ell; r]$ be the last interval considered before halting. According to the majority of the elements in $I[\ell-2\delta;\ell]$, x is either contained in $I[\ell+1;n]$ or not contained in I. This is true since the mentioned majority contains at least $(\delta+1)$ elements, and thus at least one of them must be faithful. A similar argument applied to $I[r; r+2\delta]$ shows that x can only be contained in $I[1;r-1]$. Since the algorithm did not find x in $I[\ell+1;n] \cap I[1;r-1] = I[\ell+1;r-1]$, there is no faithful key equal to x in I.

Now consider the time spent in one iteration of the algorithm (starting from the initial interval $I = I[1;n]$). Each time the algorithm selects a random element, either the algorithm halts or the size of the subsequence considered is decreased by at least a factor of $2/(1+c) > 1$. So the total number of selection steps is $O(\log n)$, where each step requires $O(1)$ time. The final step, where a subsequence of length at most $4\delta + C\delta = O(\delta)$ is considered, requires $O(\delta)$ time. Altogether, the worst-case time for one iteration is $O(\log n + \delta)$.

Thus, it is sufficient to show that in a given iteration the algorithm halts (that is, it either finds x or answers no) with some positive constant probability $P > 0$, from which it follows that the expected number of iterations is constant. Let $I[h_1], I[h_2] \ldots I[h_t]$ be the sequence of randomly chosen values in a given iteration. If a new iteration starts, this implies that at least one of those values is faulty. Hence, to show that the algorithm halts, it is sufficient to prove that all those values are faithful with positive probability.

Let $\overline{P}_k$ denote the probability that $I[h_k]$ is faulty. Consider the last interval $I[\ell; r]$ in which we perform random sampling. The length of this interval is at least $C\,\delta$. So the value $I[h_t]$ is chosen in a subsequence of length at least $c\,C\,\delta > \delta$, from which we obtain $\overline{P}_t \leq \delta/(c\,C\,\delta) = 1/(c\,C)$. Consider now the previous interval. The length of this interval is at least $2C\delta/(1+c)$. Thus $\overline{P}_{t-1} \leq (1+c)/(2\,c\,C)$. More generally, for each $i = 0, 1, \ldots (t-1)$, we have $\overline{P}_{t-i} \leq ((1+c)/2)^i / (cC)$. Altogether, the probability P that all the values $I[h_1], I[h_2] \ldots I[h_t]$ are faithful is equal to $\prod_{i=0}^{t-1}(1-\overline{P}_{t-i})$ and thus

$$P \geq \prod_{i=0}^{t-1}\left(1 - \frac{1}{c\,C}\left(\frac{1+c}{2}\right)^i\right) \geq \left(1 - \frac{1}{c\,C}\right)^{\sum_{i=0}^{t-1}(\frac{1+c}{2})^i} \geq \left(1 - \frac{1}{c\,C}\right)^{\frac{2}{1-c}} > 0,$$

where we used the fact that $(1-xy) \geq (1-x)^y$ for every x and y in $[0,1]$.

Almost Optimal Deterministic Searching. Before describing our deterministic algorithm, which we refer to as `DetSearch`, we introduce the notion of *k-left-test* and *k-right-test* over a position p, for $k \geq 1$ and $1 \leq p \leq n$. In a k-left-test over p, we consider the neighborhood of p of size k defined as $I[p-k\,;\,p-1]$: the test *fails* if the majority of keys in this neighborhood is larger than the key x to be searched for, and *succeeds* otherwise. A k-right-test over p is defined symmetrically on the neighborhood $I[p+1\,;\,p+k]$. Note that in the randomized searching algorithm described in the previous section we execute a $(2\delta+1)$-left-test and a $(2\delta+1)$-right-test at the end of each iteration. The idea behind our improved deterministic algorithm is to design less expensive left and right tests, and to perform them more frequently.

More precisely, the basic structure of the algorithm is as in the classical (deterministic) binary search: in each step we consider the current interval $I[\ell;r]$ and we update it as suggested by the central value $I[(\ell+r)/2]$. Every $\sqrt{\delta}$ searching steps, we perform a $\sqrt{\delta}$-left-test over the left boundary ℓ and a $\sqrt{\delta}$-right-test over the right boundary r of the current interval $I[\ell;r]$. If one of the two $\sqrt{\delta}$-tests fails, we revert to the smallest interval $I[\ell';r']$ suggested by the failed test and by the last $\sqrt{\delta}$-tests previously performed (the boundaries ℓ' and r' can be maintained in safe memory, and are updated each time a $\sqrt{\delta}$-test is performed). Every δ searching steps, we proceed analogously, where $\sqrt{\delta}$-tests are replaced by $(2\delta+1)$ tests.

We defer the low-level details, the description of the boundary cases, and the proof of correctness of algorithm `DetSearch` to the full paper. We now analyze the running time. We will say that a boundary p is *misleading* if the value $I[p]$ is faulty and guides the search towards a wrong direction. Similarly, a k-left-test over p is misleading if the majority of the values in $I[p-k\,;\,p-1]$ are misleading.

Theorem 5. *Algorithm* `DetSearch` *performs resilient searching in* $O(\log n + \alpha\sqrt{\delta}\,)$ *worst-case time.*

Proof. Assume that the algorithm takes at some point a wrong search direction (*mislead search*). We first analyze the running time for a mislead search when there is no misleading $\sqrt{\delta}$-test. Without loss of generality, consider the case where the algorithm encounters a misleading left boundary, say p: then, the search erroneously proceeds to the right of p. Consider the time when the next $\sqrt{\delta}$-left-test is performed, and let ℓ be the left boundary involved in the test. Note that it must be $p \leq \ell$ and, since p is misleading, then ℓ must be also a misleading left boundary. Due to the hypothesis that $\sqrt{\delta}$-tests are not misleading, the $\sqrt{\delta}$-left-test over ℓ must have failed, detecting the error on p and recovering the proper search direction: hence, the uncorrect search wasted only $O(\sqrt{\delta}\,)$ time, which can be charged to the faulty value $I[\ell]$. Since $I[\ell]$ is out of the interval on which the search proceeds, each faulty value can be charged at most once and we will have at most α uncorrect searches of this kind. The total running time will thus be $O(\alpha\sqrt{\delta}\,)$.

We next analyze the running time for a mislead search when there exists at least one misleading $\sqrt{\delta}$-test. In this case, an error due to a misleading $\sqrt{\delta}$-test will be detected at most δ steps later, when the next $(2\delta+1)$-test is performed. Using similar arguments, we can prove that there must exist $\Theta(\sqrt{\delta}\,)$ faulty values that are eliminated from the interval in which the search proceeds, and we can charge the $O(\delta)$ time spent for the uncorrect search to those values. Thus, we will have at most $O(\alpha/\sqrt{\delta}\,)$ uncorrect searches of this kind, requiring $O(\delta)$ time each. The total running time will be again $O(\alpha\sqrt{\delta}\,)$. Since the time for the correct searches is $O(\log n)$, the claimed bound of $O(\log n + \alpha\sqrt{\delta}\,)$ follows.

The running time can be reduced to $O(\log n + \alpha\,\delta^{\epsilon'})$, for any constant $\epsilon' \in (0,1]$, by exploiting the use of $(2\delta^{i\,\epsilon'}+1)$-tests, with $i = 1, 2, \ldots (1/\epsilon')$. This yields a deterministic resilient searching algorithm that can tolerate up to $O((\log n)^{1-\epsilon})$ memory faults, for any small positive constant ϵ, in $O(\log n)$ worst-case time, thus getting arbitrarily close to the lower bound. We omit here the details of the algorithm for lack of space.

Acknowledgments. This work has been partially supported by the Sixth Framework Programme of the EU under Contract Number 507613 (Network of Excellence "EuroNGI: Designing and Engineering of the Next Generation Internet") and by MIUR, the Italian Ministry of Education, University and Research, under Projects WEB MINDS ("Wide scalE Broadband MIddleware for Networks Distributed Services") and ALGO-NEXT ("Algorithms for the Next Generation Internet and Web: Methodologies, Design and Experiments").

References

1. J. A. Aslam and A. Dhagat. Searching in the presence of linearly bounded errors. *Proc. 23rd ACM Symp. on Theory of Computing* (STOC'91), 486–493, 1991.
2. S. Assaf and E. Upfal. Fault-tolerant sorting networks. *SIAM J. Discrete Math.*, 4(4), 472–480, 1991.
3. Y. Aumann and M. A. Bender. Fault-tolerant data structures. *Proc. 37th IEEE Symp. on Foundations of Computer Science* (FOCS'96), 580–589, 1996.
4. J. Blömer and J.-P. Seifert. Fault based cryptanalysis of the Advanced Encryption Standard (AES). *Proc. 7th International Conference on Financial Cryptography* (FC'03), LNCS 2742, 162–181, 2003.
5. R. S. Borgstrom and S. Rao Kosaraju. Comparison based search in the presence of errors. *Proc. 25th ACM Symp. on Theory of Computing* (STOC'93), 130–136, 1993.
6. R. Boyer and S. Moore. MJRTY - A fast majority vote algorithm. University of Texas Tech. Report, 1982.
7. B. S. Chlebus, A. Gambin and P. Indyk. Shared-memory simulations on a faulty-memory DMM. *Proc. 23rd International Colloquium on Automata, Languages and Programming* (ICALP'96), 586–597, 1996.
8. B. S. Chlebus, L. Gasieniec and A. Pelc. Deterministic computations on a PRAM with static processor and memory faults. *Fundamenta Informaticae*, 55(3-4), 285–306, 2003.
9. M. Farach-Colton. Personal communication. January 2002.
10. U. Feige, P. Raghavan, D. Peleg, and E. Upfal. Computing with noisy information. *SIAM Journal on Computing*, 23, 1001–1018, 1994.
11. I. Finocchi and G. F. Italiano. Sorting and searching in the presence of memory faults (without redundancy). *Proc. 36th ACM Symposium on Theory of Computing* (STOC'04), 101–110, 2004.
12. S. Hamdioui, Z. Al-Ars, J. Van de Goor, and M. Rodgers. Dynamic faults in Random-Access-Memories: Concept, faults models and tests. *Journal of Electronic Testing: Theory and Applications*, 19, 195–205, 2003.
13. M. Henzinger. The past, present and future of Web Search Engines. Invited talk. *31st Int. Coll. Automata, Languages and Programming*, Turku, Finland, July 12–16 2004.
14. P. Indyk. On word-level parallelism in fault-tolerant computing. *Proc. 13th Annual Symp. on Theoretical Aspects of Computer Science* (STACS'96), 193–204, 1996.
15. D. J. Kleitman, A. R. Meyer, R. L. Rivest, J. Spencer, and K. Winklmann. Coping with errors in binary search procedures. *Journal of Computer and System Sciences*, 20:396–404, 1980.
16. K. B. Lakshmanan, B. Ravikumar, and K. Ganesan. Coping with erroneous information while sorting. *IEEE Trans. on Computers*, 40(9):1081–1084, 1991.
17. T. Leighton and Y. Ma. Tight bounds on the size of fault-tolerant merging and sorting networks with destructive faults. *SIAM Journal on Computing*, 29(1):258–273, 1999.

18. T. Leighton, Y. Ma and C. G. Plaxton. Breaking the $\Theta(n \log^2 n)$ barrier for sorting with faults. *Journal of Computer and System Sciences*, 54:265–304, 1997.
19. T. C. May and M. H. Woods. Alpha-Particle-Induced Soft Errors In Dynamic Memories. *IEEE Trans. Elect. Dev.*, 26(2), 1979.
20. S. Muthukrishnan. On optimal strategies for searching in the presence of errors. *Proc. 5th ACM-SIAM Symp. on Discrete Algorithms* (SODA'94), 680–689, 1994.
21. A. Pelc. Searching games with errors: Fifty years of coping with liars. *Theoretical Computer Science*, 270, 71–109, 2002.
22. B. Ravikumar. A fault-tolerant merge sorting algorithm. *Proc. 8th Annual Int. Conf. on Computing and Combinatorics* (COCOON'02), LNCS 2387, 440–447, 2002.
23. A. Rényi. *A diary on information theory*, J. Wiley and Sons, 1994. Original publication: *Napló az információelméletröl*, Gondolat, Budapest, 1976.
24. S. Skorobogatov and R. Anderson. Optical fault induction attacks. *Proc. 4th Int. Workshop on Cryptographic Hardware and Embedded Systems*, LNCS 2523, 2–12, 2002.
25. Tezzaron Semiconductor. *Soft errors in electronic memory - a white paper*, URL: `http://www.tezzaron.com/about/papers/Papers.htm`, January 2004.
26. S. M. Ulam. *Adventures of a mathematician.* Scribners (New York), 1977.
27. A. C. Yao and F. F. Yao. On fault-tolerant networks for sorting. *SIAM Journal on Computing*, 14, 120–128, 1985.

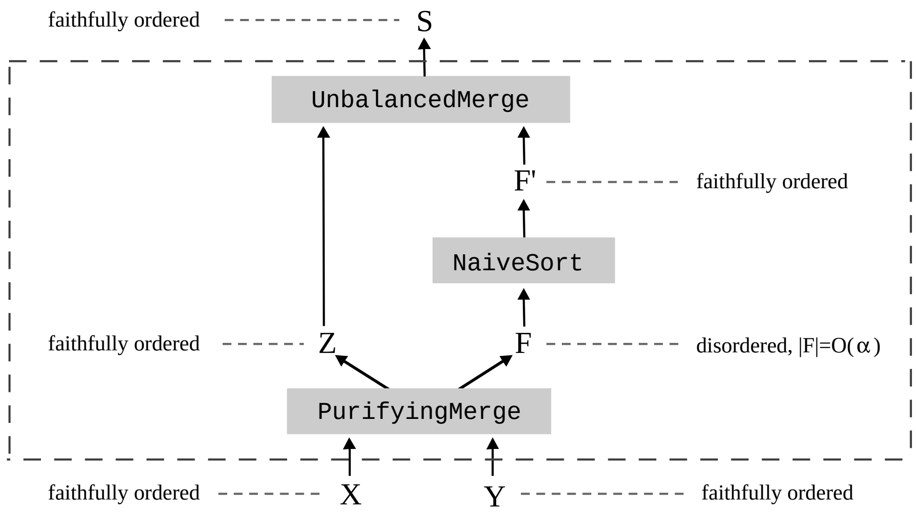

Fig. 1. Our resilient merging algorithm.