Designing Reliable Algorithms in Unreliable Memories*

Irene Finocchi † Fabrizio Grandoni † Giuseppe F. Italiano ‡

Abstract

Some of today's applications run on computer platforms with large and inexpensive memories, which are also error-prone. Unfortunately, the appearance of even very few memory faults may jeopardize the correctness of the computational results. We say that an algorithm is resilient to memory faults if, despite the corruption of some memory values before or during its execution, it is nevertheless able to get a correct output at least on the set of uncorrupted values (i.e., the algorithm works correctly on uncorrupted data). In this paper we will survey some recent work on resilient algorithms and try to give some insight on the main algorithmic techniques used.

Keywords: memory faults, unreliable information, combinatorial algorithms, resilient sorting and searching, resilient data structures.

1 Introduction

Many large-scale applications require the processing of massive data sets, which can easily reach the order of Terabytes. For instance, NASA's Earth Observing System generates Petabytes of data per year, while Google currently reports to be indexing and searching over 8 billion Web pages. In all such applications processing massive data sets, there is an increasing demand for large, fast, and inexpensive memories, at any level of the memory hierarchy: this trend is witnessed, e.g., by the large popularity achieved in recent years by commercial Redundant Arrays of Inexpensive Disks (RAID) systems [14, 31], which offer enhanced I/O bandwidths, large capacities, and low cost. Memories used in today's computer platforms, especially cheap ones, are not fully safe, and sometimes the content of a memory unit may be temporarily or permanently lost or damaged. This may depend on

*Work supported in part by the Italian MIUR Project MAINSTREAM: "Algorithms for Massive Information Structures and Data Streams".

†Dipartimento di Informatica, Università di Roma "La Sapienza", via Salaria 113, 00198, Roma, Italy. Email: {finocchi,grandoni}@di.uniroma1.it.

‡Dipartimento di Informatica, Sistemi e Produzione, Università di Roma "Tor Vergata", via del Politecnico 1, 00133 Roma, Italy. Email: italiano@disp.uniroma2.it.

manufacturing defects, power failures, or environmental conditions such as cosmic radiation and alpha particles [26, 36, 43, 45]. As the memory size becomes larger, the mean time between failures decreases considerably: assuming standard soft error rates for the internal memories currently on the market [43], a system with Terabytes of memory (e.g., a cluster of computers with a few Gigabytes per node) would experience one bit error every few minutes.

A faulty memory may cause a variety of problems in most software applications, which in some cases can also pose very serious threats, such as breaking cryptographic protocols [7, 9, 47], taking control over a Java Virtual Machine [25] or breaking smart cards and other security processors [1, 2, 42]. Most of these fault-based attacks 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. Differently from the almost random errors affecting the behavior of large size memories, in this context the errors are introduced by a malicious adversary, which can assume some knowledge of the algorithm's behavior.

We stress that even very few memory faults may jeopardize the correctness of the underlying algorithms. Consider for example the following simplified scenario. We are given a set of n Web pages, each one with its own rank, and we wish to sort them by rank using a classical sorting algorithm, say, `MergeSort`. It is easy to see that the corruption of a single value of the rank is sufficient to make the algorithm output a sequence with $\Theta(n)$ Web pages in a wrong position. Similar phenomena have been observed in practice [27, 28].

Error checking and correction circuitry added at the board level could contrast memory faults, but would also impose non-negligible costs in performance and money: hence, it is not a feasible solution when speed and cost are both at prime concern. In the design of reliable systems, when specific hardware for fault detection and correction is not available or is too expensive, it makes sense to look for a solution to these problems at the application level, i.e., to design algorithms and data structures that are able to perform the tasks they were designed for, even in the presence of unreliable or corrupted information.

Suppose we are given an upper bound δ on the number of faults that can happen during the lifetime of an algorithm/data structure. Then a trivial approach to protect the computation against memory faults would be data replication, in combination with majority techniques. For example, in the scenario above we might maintain $2\delta + 1$ copies of each rank value, and use the majority value (which must be correct) as the actual rank value during the execution of the sorting algorithm. This way, all the Web pages would be sorted correctly. However, the resulting algorithm would take $O(\delta\, n \log n)$ time and $O(\delta\, n)$ space. In most applications this overhead is not acceptable, even for moderate values of δ. On the other side, it is not clear how to avoid this overhead, especially if we insist on computing a totally correct output (even on corrupted data).

The situation changes dramatically if we relax the definition of correctness, by requiring that algorithms/data structures are consistent only with respect to the uncorrupted data. For example, we might accept that the Web pages whose rank gets corrupted possibly occupy a wrong position in the output sequence, provided that at least all the remaining

Web pages are sorted correctly. As we will show, this relaxed problem can be solved in $O(n \log n + \delta^2)$ time and $O(n)$ space, which is asymptotically optimal for $\delta = O(\sqrt{n \log n})$. This algorithm also performs very well in practice (see Section 3.1).

As we will see, many fundamental problems in the design of algorithms and data structures, such as the sorting problem mentioned before, have a natural and meaningful resilient formulation, where the correctness constraint is properly relaxed. In this paper we survey some recent work on the design and analysis of *resilient* algorithms and data structures, that is algorithms and data structures which solve the resilient version of classical problems.

A model for memory faults. In this survey we focus on the *faulty-memory RAM* model for memory faults, which was introduced in [24]. In this model, it is assumed that there is an adaptive adversary which can corrupt up to δ memory words, in any place and at any time (even simultaneously). We remark that δ is not a constant, but a parameter of the model. The adaptive adversary captures situations like cosmic-rays bursts, memories with non-uniform fault-probability, and hackers' attacks which would be difficult to be modelled otherwise.

The model also assumes that there are $O(1)$ safe memory words which cannot be accessed by the adversary. This memory stores the code of the algorithm/data structure (which otherwise could be corrupted by the adversary), together with a small number of running variables. This assumption is not very restrictive, since typically the space occupied by the code is orders of magnitude smaller than the space taken by data: hence one can usually afford the cost of storing the code in a smaller, more expensive and more reliable memory. We remark that a constant-size reliable memory may even not be sufficient for a recursive algorithm to work properly: parameters, local variables, return addresses in the recursion stack may indeed get corrupted.

In the randomized version of the model [23] the random bits are not accessible to the adversary. Moreover, reading a memory word (in the unsafe memory) is an atomic operation, that is the adversary cannot corrupt a memory word after the reading process has started. Without the last two assumptions, most of the power of randomization would be lost in the setting considered.

In the following, we will denote by α the *actual* number of faults that happen during a specific execution of an algorithm or data structure. Note that $\alpha \leq \delta$.

Organization. The remainder of this paper is organized as follows. In Section 2 we survey some alternative models of faulty environments considered in the literature, and compare them with the faulty-memory RAM. In Section 3 we consider the resilient sorting problem, both in the comparison-based setting (Section 3.1) and in the case of integer values (Section 3.2). Section 4 is devoted to resilient searching in a sorted sequence. In Section 5 we focus on the design and analysis of resilient data structures. In particular, we consider search trees (Section 5.1) and priority queues (Section 5.2). We conclude with some remarks and open problems in Section 6.

2 Related Models

The problem of computing with unreliable information or in the presence of faulty components dates back to the 50's [46]. Due to the heterogeneous nature of faults (e.g., permanent or transient) and to the large variety of components that may be faulty in computer platforms (e.g., processors, memories, network nodes or links), many different models have been proposed in the literature. We next briefly survey a few such models, and relate them to the faulty-memory RAM.

The liar model. Two-person games in the presence of unreliable information have been the subject of extensive research since Rényi [41] and Ulam [44] posed the following "twenty questions problem":

> Two players, Paul and Carole, are playing the game. Carole thinks of a number between one and one million, which Paul has to guess by asking up to twenty questions with binary answers. How many questions does Paul need to get the right answer if Carole is allowed to lie once or twice?

Many subsequent works have addressed the problem of searching by asking questions answered by a possibly lying adversary [4, 10, 19, 20, 32, 37, 38, 39]. These works consider questions of different formats (e.g., comparison questions or general yes-no questions such as "Does the number belong to a given subset of the search space?") and different degrees of interactivity between the players (in the adaptive framework, Carole must answer each question before Paul asks the next one, while in the non-adaptive framework all questions must be issued in one batch). We remark that the problem of finding optimal searching strategies has strong relationships with the theory of error correcting codes. Furthermore, different kinds of limitations can be posed on the way Carole is allowed to lie: e.g., fixed number of errors, probabilistic error model, or linearly bounded model in which the number of lies can grow proportionally with the number of questions. Even in the very difficult linearly bounded model, searching is now well understood and can be solved to optimality: Borgstrom and Kosaraju [10], improving over [4, 19, 38], designed an $O(\log n)$ searching algorithm. We refer to the excellent survey by Pelc [39] for an extensive bibliography on this topic.

Problems such as sorting and selection have instead substantially different bounds. Lakshmanan *et al.* [33] proved that $\Omega(n \log n + k \cdot n)$ comparisons are necessary for sorting when at most k lies are allowed. The best known $O(n \log n)$ algorithm tolerates only $O(\log n / \log \log n)$ lies, as shown by Ravikumar in [40]. In the linearly bounded model, an exponential number of questions is necessary even to test whether a list is sorted [10]. Feige *et al.* [20] studied a probabilistic model and presented a sorting algorithm correct with probability at least $(1 - q)$ that requires $\Theta(n \log(n/q))$ comparisons. Lies are well suited at modelling transient ALU failures, such as comparator failures.

We remark that, since memory data get never corrupted in the liar model, fault-tolerant algorithms may exploit query replication strategies. This kind of approach does not work in the faulty-memory RAM model.

Fault-tolerant sorting networks. Destructive faults have been first investigated in the context of fault-tolerant sorting networks [5, 34, 35, 48], in which comparators can be faulty and can possibly destroy one of the input values. Assaf and Upfal [5] present an $O(n \log^2 n)$-size sorting network tolerant (with high probability) to random destructive faults. Later, Leighton and Ma [34] proved that this bound is tight. The Assaf-Upfal network makes $\Theta(\log n)$ copies of each item, using data replicators that are assumed to be fault-free. We observe that, differently from fault-tolerant sorting networks, the algorithms and data structures that we are going to present use data replication only at a very limited extent, i.e., without increasing asymptotically the space consumption with respect to their non-resilient counterparts.

Parallel computing with memory faults. Multiprocessor platforms are even more prone to hardware failures than uniprocessor computers. A lot of research has been thus devoted to deliver general simulation mechanisms of fully operational parallel machines on their faulty counterparts. The simulations designed in [15, 16, 17, 29] are either randomized or deterministic, and operate with constant or logarithmic slowdown, depending on the model (PRAM or Distributed Memory Machine), on the nature of faults (static or dynamic, deterministic or random), and on the number of available processors. Some of these works also assume the existence of a special fault-detection register that makes it possible to recognize memory errors upon access to a memory location. This is different from the faulty-memory RAM model, where corrupted values are not distinguishable from correct ones.

Pointer-based data structures. Unfortunately, many pointer-based data structures are highly non-resilient: losing a single pointer makes the entire data structure unreachable. This problem has been addressed in [6], providing fault-tolerant versions of stacks, linked lists, and binary search trees: these data structures have a small time and space overhead with respect to their non-fault-tolerant counterparts, and guarantee that only a limited amount of data may be lost upon the occurrence of memory faults. In contrast, in the faulty-memory RAM model correct values cannot get lost.

Checkers. Blum et al. [8] considered the following problem: given a data structure residing in a large unreliable memory controlled by an adversary and a sequence of operations that the user has to perform on the data structure, design a checker that is able to detect any error in the behavior of the data structure while performing the user's operations. The checker can use only a small amount of reliable memory and can report a buggy behavior either immediately after an errant operation (on-line checker) or at the end of the sequence (off-line checker). Memory checkers for random access memories, stacks and queues have been presented in [8], where lower bounds of $\Omega(\log n)$ on the amount of reliable memory needed in order to check a data structure of size n are also given.

It is worth to mention that the algorithms and data structures that we are going to present might not always be able to detect faulty behaviors: nonetheless, they operate

correctly on the uncorrupted data.

3 Resilient Sorting

In this section we consider the problem of sorting in the presence of memory faults, both in the comparison-based setting and in the case of integer values. Recall that we do not wish to recover corrupted data, but simply to be correct on uncorrupted data, with a low time and space overhead. Hence, we can define the following natural resilient version of the sorting problem:

> *Resilient sorting*: we are given a set of n keys that need to be sorted. The values of some keys may be arbitrarily corrupted during the sorting process. The problem is to order correctly the set of uncorrupted keys.

Note that this is the best that we can achieve in the presence of memory faults: we cannot indeed prevent keys corrupted at the very end of the algorithm execution from occupying wrong positions in the output sequence.

We remark that one of the main difficulties in designing efficient resilient sorting algorithms derives from the fact that positional information is no longer reliable in the presence of memory faults. Consider for instance the classical `MergeSort` algorithm: during the merge step, errors may propagate due to corrupted keys. Even in the presence of a single fault, in the worst case as many as $\Theta(n)$ keys may be out of order in the output sequence.

Observe that, if each value were replicated $2\delta + 1$ times, by majority techniques we could easily tolerate up to δ faults; however, the algorithm would present a multiplicative overhead of $\Theta(\delta)$ in terms of both space and running time. This implies, for instance, that in order to be resilient to $O(\sqrt{n})$ faults, a sorting algorithm would require $\Omega(n^{3/2} \log n)$ time and $\Omega(n^{3/2})$ space. The space may be improved using error-correcting codes, but at the price of a higher running time. We will see in this section that it is possible to do much better.

In the following, we will assume that the input keys are initially contained into an array, whose address and length are maintained in safe memory. Otherwise, uncorrupted keys might be lost because of pointers corruption. For ease of presentation, we will also assume that each key occupies one memory location. This second constraint can be partially relaxed, but that would not add much to the discussion below.

3.1 Comparison-Based Sorting

In [23, 24] both upper and lower bounds for the resilient comparison-based sorting problem are presented. Any deterministic algorithm with optimal running time $O(n \log n)$ can tolerate the corruption of at most $O(\sqrt{n \log n})$ keys. This lower bound implies that, if we have to sort in the presence of $\omega(\sqrt{n \log n})$ memory faults, then we should be prepared to spend more than $O(n \log n)$ time. In [24] a first resilient $O(n \log n)$ comparison-based

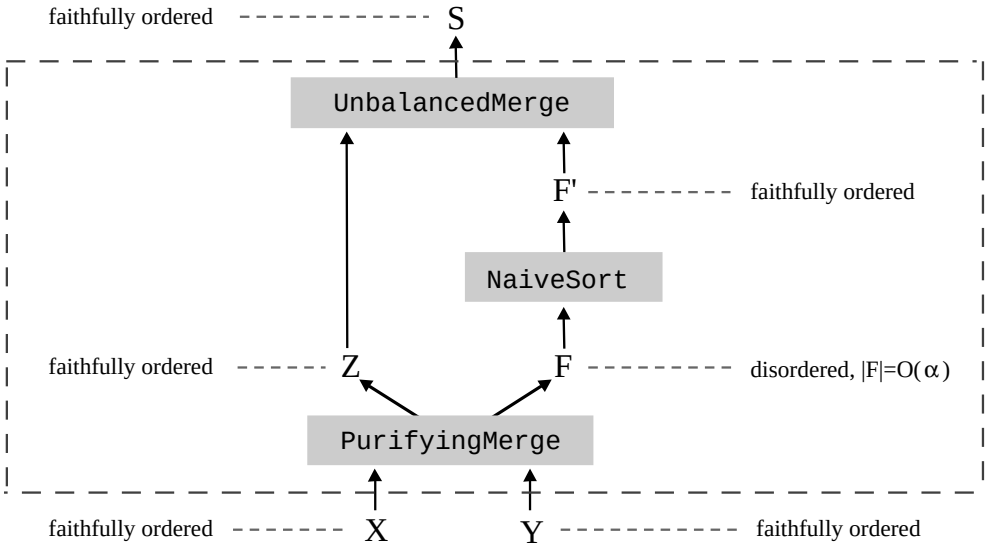

Figure 1: The resilient merging procedure.

sorting algorithm that tolerates $O((n \log n)^{1/3})$ memory faults has been designed. The gap between the upper bound and the lower bound has been later closed in [23], where a resilient sorting algorithm that takes $O(n \log n + \delta^2)$ worst-case time is presented. This yields an algorithm that can optimally tolerate up to $O(\sqrt{n \log n})$ faults in $O(n \log n)$ time.

The comparison-based sorting algorithms of [23, 24] are based on the same basic approach. The idea is to use a bottom-up iterative implementation of `MergeSort`, with a carefully adapted (resilient) merging procedure. A sequence of keys is *faithfully sorted* if the subsequence induced by the uncorrupted keys is sorted. Given two faithfully sorted input sequences X and Y of size k, the aim is to merge them into a unique faithfully sorted sequence S. The resilient merging procedure hinges upon the combination of two merging subroutines, `PurifyingMerge` and `UnbalancedMerge`, with quite different characteristics. The first subroutine is fast, but may discard a subset F of keys. The keys F are faithfully sorted with a trivial algorithm `NaiveSort`, taking time $O(\delta|F|)$, and then merged with the remaining keys Z, in time $O(|Z| + (|F| + \alpha)\delta)$, by using the second subroutine, which is slow in general, but is especially efficient when applied to unbalanced sequences. (See also Figure 1).

The algorithms in [24] and [23] differ in the implementation of `PurifyingMerge`: in the former case the subroutine requires $O(k)$ time and discards $O(\alpha\delta)$ elements, while in the latter case it requires $O(k+\alpha\delta)$ time but discards only $O(\alpha)$ elements. These bounds yield a total running time $O(k + \alpha\delta^2)$ and $O(k + \alpha\delta)$ for resilient merging. This in turn yields two resilient implementations of `MergeSort` that run in $O(n \log n + \delta^3)$ and $O(n \log n + \delta^2)$ time, respectively.

The `PurifyingMerge` procedure in [23] is simple enough to be sketched here. The basic idea is to use two input buffers $\mathcal{X}$ and $\mathcal{Y}$, and one output buffer $\mathcal{Z}$, all of size $\Theta(\delta)$. The top values of X (Y) are initially moved into $\mathcal{X}$ ($\mathcal{Y}$). Then the content of $\mathcal{X}$ and $\mathcal{Y}$ is

merged into $\mathcal{Z}$ in the standard way. At the end of the process, the algorithm performs a consistency check, of cost $O(\delta)$. If the check succeeds, the content of $\mathcal{Z}$ is appended to the output sequence Z. Otherwise, the algorithm is able to identify a pair of keys which are not ordered correctly in either $\mathcal{X}$ or $\mathcal{Y}$: these two keys are moved to the fail set F, and the merging process is restarted from scratch on the remaining keys. Since at least one of the two removed keys must be corrupted (being the input sequences faithfully sorted by assumption), at the end of the process $|F| \leq 2\alpha$.

The overall cost of the successful merging steps is $O(k)$. We can charge the $O(\delta)$ cost of each unsuccessful merging step on the corresponding corrupted key which is removed from the input sequences. This gives the claimed $O(k + \alpha\,\delta)$ running time.

In [21] the impact of memory faults both on the correctness and on the running times of `MergeSort`-based sorting algorithms, including the resilient algorithms presented in [23, 24], has been experimentally investigated. In particular, many experiments with a variety of fault injection strategies and different instance families have been carried out. The experiments performed give evidence that simple-minded approaches to the resilient sorting problem are largely impractical, while the design of more sophisticated algorithms seems really worth the effort: the algorithms presented in [23, 24] are not only theoretically efficient, but also fast in practice. In particular, an engineered implementation of [23] introduced in [21] is robust to different memory fault patterns and appears to be the algorithm of choice for most parameter choices. The algorithm of [24], however, seems to have smaller constants hidden in the asymptotic notation and might be preferable for rather small values of δ.

3.2 Integer Sorting

Suppose the input sequence is formed by integers in the range $[0, n^c - 1]$, for some constant $c \geq 0$. In [23] a randomized sorting algorithm with expected running time $O(n + \delta^2)$ is presented: thus, this algorithm is able to tolerate up to $O(\sqrt{n}\,)$ memory faults in expected linear time. The same task can be performed deterministically in time $O(n + \delta^{2+\epsilon})$, for any constant $\epsilon > 0$. No lower bound for resilient integer sorting is currently known.

The integer sorting algorithm in [23] is based on a resilient implementation of `RadixSort`. If the integers are represented in base b, where $b \geq 2$ is any constant, `RadixSort` can be easily implemented in faulty memory by keeping b buckets, each stored in an array of size n, and maintaining the addresses of the arrays and their current lengths in the $O(1)$-size safe memory. However, when b is constant the running time of `RadixSort` is $O(n \log n)$. In order to make `RadixSort` run in linear time, we need $b = \Omega(n^\epsilon)$, for some constant $\epsilon \in (0, 1]$. This implies that $O(1)$ safe memory words are not sufficient to store the initial addresses and the current lengths of the b buckets. While the problem with the addresses can be solved by storing the arrays in a proper way, the problem with the lengths is more serious. We therefore need to solve b instances of a *bucket-filling* problem defined as follows: We receive in an online fashion a sequence of k 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. Array $\mathcal{B}_0$ must maintain the same faithful order

as the order in the input sequence.

The bucket-filling problem can be solved in $O(k+\alpha\delta)$ expected time, and in $O(k+\alpha\delta^{1+\epsilon})$ deterministic time. This gives the claimed $O(n+\delta^2)$ expected and $O(n+\delta^{2+\epsilon})$ worst-case running times. To give an intuition of the techniques used to solve the bucket-filling problem, we next sketch a simpler algorithm performing the task in $O(k+\alpha\delta^{1.5})$ worst-case time. 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. The current length p_i of $\mathcal{B}_i$ is kept in unsafe memory in multiple copies. Specifically, we maintain one copy of p_2, $|\mathcal{B}_2|$ copies of p_1, and $|\mathcal{B}_1|$ copies of p_0. The majority value of the copies of p_i is interpreted as the actual value of p_i. Note that the adversary may corrupt the value of p_2 and p_1 (not of p_0). However, doing that requires $\Omega(1)$ and $\Omega(\sqrt{\delta})$ faults, respectively. 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 p_2), we *flush* it as follows: (1) we remove any $+\infty$ from $\mathcal{B}_2$ and sort $\mathcal{B}_2$ with two-way `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).

To show the correctness, we notice that all the faithful keys eventually appear in $\mathcal{B}_0$. All the faithful keys in $\mathcal{B}_i$, $i \in \{1,2\}$, at a given time precede the faithful keys not yet copied into $\mathcal{B}_i$. Moreover we sort $\mathcal{B}_i$ before flushing it. This guarantees that the faithful keys are moved from $\mathcal{B}_i$ to $\mathcal{B}_{i-1}$ in a first-in-first-out fashion. `Bubblesort` has the property to faithfully sort a k-unordered sequence of n keys in time $O(n+(k+\alpha)n)$. 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 ($\alpha'' \leq \sqrt{\delta}$), 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 $O(k + (\alpha/\sqrt{\delta})\,\delta^2 + \alpha\,\delta) = O(k + \alpha\,\delta^{1.5})$. Using a similar argument, it can be shown that the total cost of flushing buffer $\mathcal{B}_2$ is $O(k + \alpha\,\delta)$. The claimed running time immediately follows.

4 Resilient Searching

Similarly to sorting, one of the main difficulties in designing efficient resilient searching algorithms derives from the fact that positional information is no longer reliable in the presence of memory faults: for instance, when we search an array whose correct keys are in increasing order, it may be still possible that a corrupted key in position i is smaller than some correct key in position j, $j < i$, thus guiding the search towards a wrong direction.

The resilient searching problem can be formalized as follows:

> *Resilient searching*: we are given a sequence of n keys sorted in increasing order, on which we wish to perform membership queries. The values of keys may be arbitrarily corrupted during the searching process. Let κ be the key to be searched for. The problem is either to find a key (corrupted or uncorrupted) equal to κ, or to determine that there is no correct key equal to κ.

It is not difficult to see that this is the best we can hope for, since memory faults can make the search-key κ appear or disappear in the sequence at any time.

Like in the case of resilient sorting, and for analogous reasons, we will assume that the input keys are contained into an array: the address and length of the array, and the search key κ are maintained in safe memory. Moreover, each key occupies one memory location.

Both upper and lower bounds for the resilient searching problem, considering both deterministic and randomized algorithms, have been proved in [12, 23, 24]. In [24] a first algorithm, taking $O(\log n + \delta^2)$ time, was designed; furthermore it was proved that any deterministic resilient searching algorithm must take $\Omega(\log n + \delta)$ time. The lower bound has been extended to randomized algorithms in [23], where an optimal $O(\log n + \delta)$ randomized searching algorithm and an almost optimal $O(\log n + \delta^{1+\epsilon})$ deterministic searching algorithm, for any constant $\epsilon > 0$, are also introduced. The gap between the upper bound and the lower bound for deterministic searching has been recently closed in [12], by giving an optimal $O(\log n + \delta)$ deterministic algorithm.

We first notice that a naive resilient searching algorithm could be easily implemented by using a majority argument at each search step, i.e., by following the search direction suggested by the majority of $(2\delta+1)$ keys in consecutive array positions. Since one can have at most δ memory faults, this guarantees that the search direction will never be wrong. However, with this approach the running time would be $O(\delta \log n)$. In order to reduce the overhead, the algorithms in [12, 23, 24] proceed in a non-resilient way, checking from time to time if they did some mistake. If this is the case, they correct the search direction.

In more detail, the algorithm in [24] works in rounds consisting of δ non-resilient binary search steps. At the end of each round it spends $\Theta(\delta)$ time to check whether the search direction is correct. If this is not the case, it recovers the search direction in $O(\delta)$ time using a simple majority argument. The running time analysis is based on two main facts. First, the $\Theta(\delta)$ time spent for the correctness check at the end of a correct round can be amortized over the δ search steps just performed. Second, the $\Theta(\delta)$ time spent during each unsuccessful round can be charged over corrupted values: since at least one corrupted value is out of the interval in which the search proceeds, each corrupted key can be charged at most once and we will have at most α unsuccessful rounds. This yields a total running time $O(\log n + \alpha\,\delta) = O(\log n + \delta^2)$. As shown in [23], the running time can be reduced to $O(\log n + \delta^{1+\epsilon})$, for any constant $\epsilon \in (0, 1)$, by performing less reliable tests more often.

The randomized algorithm in [23] is based on the following intuition: if δ is much smaller than n and at each search step we compare the key κ to be searched for against a randomly chosen key (instead of the central one), the probability of sampling a corrupted

key and hence of possibly going towards a wrong direction will be small. This probability, however, becomes larger and larger as the search proceeds, because the size of the interval in which κ must be searched for is progressively reduced. Hence, when the interval is too small (i.e., contains less than $c \cdot \delta$ keys for some constant $c > 1$), the algorithm switches to an exhaustive search. Since corrupted sampled keys can mislead the search, at the end of the process the algorithm checks, in $O(\delta)$ time, whether the search proceeded in the right way. If not, a new search is performed from scratch. One such iteration requires $O(\log n + \delta)$ time. It can be shown that, with constant positive probability, in a given iteration no corrupted value is sampled, and hence the algorithm halts. This yields the desired result.

The optimal deterministic algorithm proposed in [12] implicitly divides the sorted input array into blocks of $5\delta + 1$ elements in consecutive positions: each block consists of a left verification segment containing the first 2δ elements, a query segment containing the next $\delta + 1$ elements, and a right verification segment containing the last 2δ elements. All the query segments implicitly define $\delta + 1$ disjoint sorted sequences $S_0, ..., S_\delta$, such that the j-th element of sequence S_i is the i-th element of the query segment of the j-th block. We remark that the elements in each sequence do not occupy consecutive positions in the input array. The algorithm stores a value $k \in [0, \delta]$ in safe memory and performs a (non-resilient) adapted binary search on the elements of sequence S_k. When the search terminates, the algorithm identifies two elements that are adjacent in S_k, and performs a verification procedure on the corresponding blocks to check whether the search was mislead by corruptions. If the verification succeeds, then the location of the search key κ must be in one of the two (consecutive) blocks: in this case all the $\Theta(\delta)$ elements in the blocks are exhaustively scanned. If the verification fails, k is incremented by one, a backtrack is performed, and the binary search continues on a different sequence S_k. Notice that since there are $\delta + 1$ disjoint sorted sequences, at least one of them must be free of faulty elements. The verification procedure that checks whether the search was mislead is a simple iterative algorithm: it maintains two values that express the confidence that the search key κ resides in the blocks on which the verification is invoked. Whenever one of the two confidence values is increased, based on simple counting arguments, a new corruption has been detected. This is crucial in the analysis, since guarantees that if $O(f)$ time is used for a verification, then $\Omega(f)$ corruptions are detected. By showing that no single corruption is counted twice, the total time spent for all the verifications can be bounded to $O(\delta)$. By charging each backtracking of the binary search to the verification procedure that triggered it, it can be finally proved that the total time of the algorithm is $O(\log n + \delta)$.

The $\Omega(\log n + \delta)$ lower bound in [23] on resilient (randomized) searching is based on the following observation. 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 κ, the first $(\delta + 1)$ values of the sequence are equal to κ 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 an uncorrupted key κ 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 one 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 in a safe-memory system, which is of course not possible.

5 Resilient Data Structures

After the design of resilient algorithms for fundamental tasks such as sorting and searching, it seems quite natural to ask whether we can successfully design resilient data structures, without incurring in extra time or space overhead. In many applications such as file system design, it is very important that the implementation of a data structure is resilient to memory faults and provides mechanisms to recover quickly from erroneous states that may lead to an incorrect behavior. The design of resilient data structures appears to be quite a challenging task. As an example, classical implementations of search trees strongly depend upon structural and positional information. Similarly to the case of binary search, the corruption of a key in a search tree may compromise the search property and guide the search towards wrong directions. Moreover, it should be noticed that pointer-based data structures are highly non-resilient: due to the corruption of one single pointer, the entire data structure may become unreachable and even uncorrupted values may be lost. A natural approach to prevent the problems described above would be to use data and pointer replication: however, if each pointer and each piece of data were replicated $\Theta(\delta)$ times, one would typically obtain a multiplicative overhead of $\Theta(\delta)$ in terms of both space and running time. In the following two subsections we will discuss results concerning resilient search trees and resilient priority queues.

5.1 Resilient Search Trees

A *resilient dictionary* is a dictionary where the search operation implements the resilient searching of Section 4, and the insert and delete operations are defined in the standard way, with respect to the mentioned search operation.

In [22] a first step towards the design of resilient data structure was made, by presenting a search tree implementing a resilient dictionary. The resilient search tree in [22] performs searches, insertions and deletions in $O(\log n + \delta^2)$ amortized time per operation, using $O(n+\delta)$ space. This implies that up to $O(\sqrt{\log n}\,)$ memory faults can be tolerated while still achieving optimal time and space bounds. For a comparison, the trivial implementation based on data replication would require $O(\delta \log n)$ time and $O(\delta n)$ space to achieve the same task. An optimal resilient dictionary has been recently proposed in [12]: the dictionary requires linear space, supports queries in $O(\log n + \delta)$ worst case time, and range queries in $O(\log n + \delta + k)$ time, where k is the size of the output. The amortized update cost is $O(\log n + \delta)$. We now sketch the main ideas behind the design of both the data structures in [22] and [12], since they are based on rather different techniques.

In the search tree described in [22], keys are grouped into a dynamically evolving set of

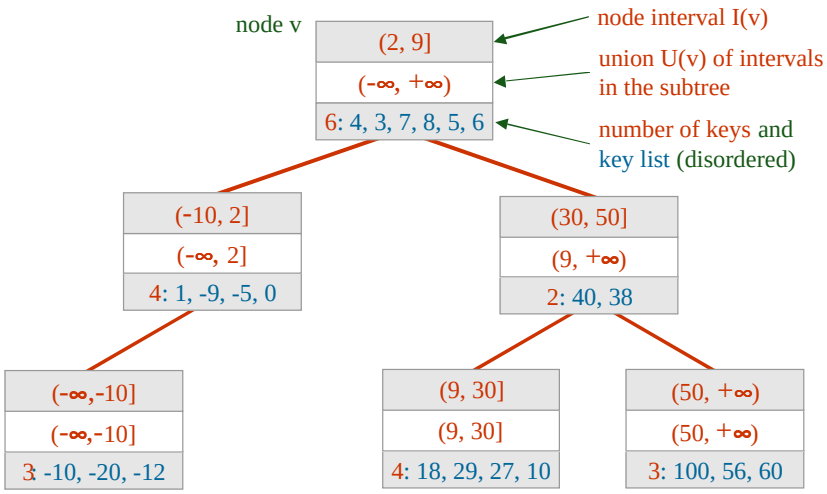

Figure 2: A resilient search tree.

non-overlapping intervals which span the key space. Intervals are maintained in a carefully adapted *AVL* tree [18]. For each node v of the search tree, one maintains the following variables:

1. (reliably, i.e., in $2\delta + 1$ copies) the endpoints of the corresponding interval $I(v)$ and the number $|I(v)|$ of keys contained in the interval $I(v)$;

2. (reliably) the interval $U(v)$ delimited by the smallest and largest endpoints of the intervals contained in the subtree rooted at v;

3. (reliably) the addresses of the left child, the right child, and the parent of v, and all the information needed to keep the search tree balanced with the implementation considered;

4. (unreliably, i.e., in a single copy) the unordered set of current keys contained in $I(v)$, stored in an array of size 2δ.

For an example, see Figure 2. The $U(v)$'s play a crucial role: they are used to test whether a search is proceeding towards the right subtree.

After insertions or deletions, intervals might be split and merged. When this happens, the search tree is modified consequently in a reliable way, in time $O(\delta \log n + \delta^2)$. By standard doubling techniques, one can ensure that: (1) each interval contains $\Theta(\delta)$ keys, and (2) the set of intervals, and thus the search tree, is modified only every $\Omega(\delta)$ insertions and/or deletions. The first property allows one to store the search tree reliably, while keeping the space linear: in fact, there are $O(n/\delta)$ intervals/nodes only, and each one can afford to use extra space $O(\delta)$ to ensure reliable computations. Thanks to the second property, the amortized cost per update is $O(\log n + \delta)$ only.

The second key ingredient of the resilient search tree in [22] lies in the way one searches for an interval containing a given key κ. This can be done in $O(\log n + \delta^2)$ worst-case

time. To give an idea of the techniques involved, we sketch how to perform the same task in $O(\log n + \alpha\delta^2) = O(\log n + \delta^3)$ worst-case time. To this aim, we generalize the checkpoint-based approach used for resilient searching that we described in Section 4. The idea is that the search algorithm must be fast on non-faulty instances, and thus it is not possible to perform reliable computations too often. On the contrary, most of the times we need to trust unreliable variables, and only every $\Theta(\delta)$ search steps we can afford to check reliably whether the computation is going towards the right direction. The search algorithm keeps track of these reliable checks by means of checkpoints: if we realize that something wrong happened, we backtrack to the last checkpoint, which is stored in safe memory. The efficiency of the algorithm hinges on the fact that searches towards wrong directions, which do not produce useful results, can be charged on corrupted values.

In more detail, the interval search proceeds in *rounds*. At the beginning of each round we are given a checkpoint node v, which initially is the search tree root, and such that $\kappa \in U(v)$. At the end of the round we find a node v' such that either $\kappa \in I(v')$ or $\kappa \in U(v') \subset U(v)$. In the latter case, node v' becomes the new checkpoint for the next round. Each round consists of at most two phases. In both phases we perform up to δ classical search steps. The difference between the two phases is that in the first one, of cost $O(\delta)$, we use only the first copy of each variable involved, while in the second one, of cost $O(\delta^2)$, we use all the $2\delta + 1$ copies of such variables. Since the first phase is not reliable, when it ends we perform a consistency check, of cost $O(\delta)$, to verify whether the search is proceeding towards the right direction. If the check fails, we start the second phase from the checkpoint v. We end the first phase and start the second phase from v also whenever we find any inconsistency.

Successful rounds, where there is no need to run the second phase, have total cost $O(\log n)$. It can be shown that one can associate a distinct fault to each unsuccessful round. This gives the claimed $O(\log n + \alpha\delta^2)$ running time.

We now describe the optimal resilient dictionary proposed in [12]. At any time during the lifetime of the data structure, the sorted sequence of elements in the dictionary is partitioned into a sequence of *leaf structures*, each storing $\Theta(\delta \log n)$ elements. Each leaf structure has a guiding element larger than all uncorrupted elements in the leaf structure itself. The $O(n/\delta \log n)$ guiding elements are placed in the leaves of a binary search tree, called *top tree*. The top tree is implemented using the (non-resilient) binary search tree described in [13], but storing all the elements and pointers reliably, i.e., using replication. As shown in [13], the top tree can be maintained such that all its levels, except for the last two, are complete. The memory lay out of the tree follows a left-to-right breadth first order, that ensures that the elements of any level are stored consecutively. Each leaf structure consists of a top bucket B and $\Theta(\log n)$ buckets: each bucket B_i contains $\Theta(\delta)$ elements, stored consecutively in memory, such that uncorrupted elements in B_i are smaller than uncorrupted elements in B_{i+1}. For each bucket B_i, the top bucket B reliably stores a guiding element larger than all elements in B_i: such elements are maintained in a sorted array. It is not difficult to see that the top tree requires $o(n)$ space, while $\Theta(n)$ space is used for all the leaf structures.

The query algorithm makes extensive use of the optimal deterministic resilient search

algorithm described in Section 4. In order to search for a key κ, the algorithm first locates two internal nodes of the top tree, say v_1 and v_2, with guiding elements g_1 and g_2 such that $g_1 \leq \kappa \leq g_2$. If h is the number of levels of the top tree, nodes v_1 and v_2 are located in level $h-3$, which is always complete. Thus, thanks to the breadth-first layout, nodes v_1 and v_2 can be identified using the deterministic resilient search algorithm described in Section 4 on the array defined by values in that level. The leaf structure possibly containing κ can be in either of the subtrees rooted at v_1 and v_2 and can be identified by performing a standard tree search in both subtrees, using the reliably stored guiding elements and pointers. The search key κ must be finally located in the appropriate leaf structure: the deterministic resilient search algorithm is used again on the top bucket B and the bucket B_i possibly containing κ is scanned exhaustively. Using the running time of the optimal deterministic resilient search algorithm, it is not difficult to see that all the operations require $O(\log n+\delta)$ time in the worst case. The breadth-first layout of the top tree also allows one to perform range queries in $O(\log n + \delta + k)$ time, where k is the size of the output.

Updates can be performed with similar techniques. Consider, as an example, the insertion of an element into the dictionary. A search is first performed in order to locate the appropriate bucket B_i in a leaf structure, the element is added to B_i and its size is updated. When the size of B_i increases too much, B_i is split into two buckets of almost equal size. The guiding elements of the two new buckets are computed and reliably inserted in the top bucket. This is done using an insertion sort step, by scanning and shifting the elements in the top bucket from right to left, and placing the new guiding element at the correct position. Similarly, when the size of the top bucket becomes too large, the top bucket is split into two new leaf structures whose guiding elements are appropriately computed and inserted into the top tree. By carefully choosing the size of buckets and top buckets, it can be shown that an update in the top tree takes $O(\delta \log^2 n)$ time, but requires $\Omega(\delta \log n)$ updates in the leaf structures: the amortized cost for operations in the top tree is therefore $O(\log n)$. Splitting a bucket of a leaf structure requires $O(\delta^2 + \delta \log n)$ time, but is done every $\Omega(\delta)$ updates, yielding an amortized cost $O(\log n + \delta)$. Similarly to the query algorithm, the initial search of the appropriate bucket in which the new key must be inserted can be also implemented in $O(\log n + \delta)$ time. The total amortized running time is therefore $O(\log n + \delta)$.

5.2 Resilient Priority Queues

A *resilient priority queue* maintains a set of elements under the operations insert and deletemin: insert adds an element and deletemin deletes and returns either the minimum uncorrupted value or a corrupted value. Observe that this definition is consistent with the resilient sorting problem discussed in Section 3: given a sequence of n elements, inserting all of them into a resilient priority queue and then performing n deletemin operations yields a sequence in which uncorrupted elements are correctly sorted.

We first notice that the resilient search tree described in Section 5.1 could be used to implement priority queues, achieving $O(\log n + \delta)$ amortized time per operation. In [30] Jorgensen *et al.* describe an alternative resilient priority queue that supports both insert

and deletemin operations withing the same time bounds, i.e., in $O(\log n + \delta)$ amortized time per operation. Thus, their priority queue matches the performance of classical optimal priority queues in the RAM model when the number of corruptions tolerated is $O(\log n)$. An essentially matching lower bound is also proved in [30]: a resilient priority queue containing n elements, with $n > \delta$, must use $\Omega(\log n + \delta)$ comparisons to answer an insert followed by a deletemin.

The resilient priority queue of [30] resembles the cache-oblivious priority queue by Arge *et al.* [3]. The main idea is to gather elements in large sorted groups, such that expensive updates do not occur too often. In more detail, the data structure consists of an insertion buffer I and of $\Theta(\log n)$ layers. Each layer L_i contains an up-buffer U_i and a down-buffer D_i: the up-buffers store large elements that are on their way up to the upper layers, whereas the down-buffers store small elements, on their way down to lower layers. Besides the internal ordering between elements in the same buffer, it is also guaranteed that all uncorrupted elements in a down buffer D_i are smaller than all uncorrupted elements in both D_{i+1} and U_{i+1}. Due to these ordering properties, elements can be moved between neighboring layers using two fundamental primitives, PUSH and PULL, that can be implemented efficiently by means of the resilient merging algorithm described in Section 3: both primitives faithfully merge consecutive buffers in the priority queue and redistribute the resulting sequence among the participating buffers. Size invariants guarantee at any time that there are not too few elements in the down buffers or too many elements in the up buffers.

New elements are added to the insertion buffer I, that is merged with the up buffer U_0 when it becomes full: if U_0 breaks the size invariant, some elements are pushed upwards by the PUSH primitive, that may be recursively invoked on the up buffers of larger layers. Due to the ordering properties, the minimum is either in the insertion buffer I or in D_0 or in U_0: if D_0 breaks the size invariant after deleting the minimum, some elements are picked from the layer above by the PULL primitive, which may be recursively invoked on the down buffers of larger layers.

It can be shown that the PUSH and PULL primitives preserve the order invariant: this implies the correctness of the implementation. The running time can be proved by amortization arguments using the size invariants.

6 Concluding Remarks

In this paper we have surveyed some recent work related to the design of algorithms and data structures resilient to memory faults. We have considered fundamental algorithmic problems, such as sorting, static and dynamic searching, and the design of data structures, such as dictionaries and priority queues. We have described upper and lower bounds achieved in the faulty memory model for the aforementioned problems, sketching the main ideas and algorithmic techniques behind the design of resilient implementations.

This preliminary work has raised many open and perhaps intriguing questions. First, the model itself may need further investigations: indeed, in the simple faulty-memory RAM of [24] there are simple problems for which it seems difficult to design efficient algorithms,

such as computing the sum of n keys. Is it possible to refine the faulty-memory model in order to take this into account? Moreover, recent work has been focusing on a faulty variant of the standard RAM model: can one design resilient algorithms for more complex memory hierarchies?

Furthermore, it may be interesting to close some of the gaps between upper and lower bounds for resilient algorithms. For instance, it would be nice to prove tight lower bounds for the resilient integer sorting problem.

All the algorithms that we have discussed in this paper make explicit use of an upper bound δ on the number of memory faults: investigating whether it is possible to obtain (efficient) resilient algorithms that do not assume any *a priori* knowledge of δ is a challenging open problem.

Finally, the design of resilient algorithms and data structures for fundamental problems appears to be important especially when processing massive data sets in large, inexpensive memories: engineering, experimentally studying the performances of these algorithms, and reducing the constant factors hidden by the asymptotic notation appear therefore to be very important for their concrete deployment in real applications.

Acknowledgments

We thank the anonymous referees for many useful comments that improved the presentation of the paper.

References

[1] R. Anderson and M. Kuhn. Tamper resistance – a cautionary note. *Proc. 2nd Usenix Workshop on Electronic Commerce*, 1–11, 1996.

[2] R. Anderson and M. Kuhn. Low cost attacks on tamper resistant devices. *Proc. International Workshop on Security Protocols*, 125–136, 1997.

[3] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. *Proc. Proc. 34th Annual ACM Symposium on Theory of Computing* (STOC'02), 268-276, 2002.

[4] 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.

[5] S. Assaf and E. Upfal. Fault-tolerant sorting networks. *SIAM J. Discrete Math.*, 4(4), 472–480, 1991.

[6] Y. Aumann and M. A. Bender. Fault-tolerant data structures. *Proc. 37th IEEE Symp. on Foundations of Computer Science* (FOCS'96), 580–589, 1996.

[7] 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.

[8] M. Blum, W. Evans, P. Gemmell, S. Kannan and M. Naor. Checking the correctness of memories. *Proc. 32th IEEE Symp. on Foundations of Computer Science* (FOCS'91), 1991.

[9] D. Boneh, R. A. DeMillo, and R. J. Lipton. On the importance of checking cryptographic protocols for faults. *Proc. EUROCRYPT*, 37–51, 1997.

[10] 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.

[11] R. Boyer and S. Moore. MJRTY - A fast majority vote algorithm. University of Texas Tech. Report, 1982.

[12] G. S. Brodal, R. Fagerberg, I. Finocchi, F. Grandoni, G. F. Italiano, A. G. Jørgensen, G. Moruz and T. Mølhave. Optimal resilient dynamic dictionaries. *Proc. 15th Annual European Symp. on Algorithms* (ESA'07), LNCS 4698, 347–358, 2007.

[13] G. S. Brodal, R. Fagerberg and R. Jacob. Cache oblivious search trees via binary trees of small height. *Proc. 13th Annual ACM-SIAM Symp. on Discrete Algorithms* (SODA'02), 39–48, 2002.

[14] P. M. Chen, E. L. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson. RAID: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26(2), 145–185, 1994.

[15] B. S. Chlebus, A. Gambin and P. Indyk. PRAM computations resilient to memory faults. *Proc. 2nd Annual European Symp. on Algorithms* (ESA'94), LNCS 855, 401–412, 1994.

[16] 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.

[17] 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.

[18] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press/McGraw-Hill Book Company, 2nd Edition, 2001.

[19] A. Dhagat, P. Gacs, and P. Winkler. On playing "twenty questions" with a liar. *Proc. 3rd ACM-SIAM Symp. on Discrete Algorithms* (SODA'92), 16–22, 1992.

[20] U. Feige, P. Raghavan, D. Peleg, and E. Upfal. Computing with noisy information. *SIAM Journal on Computing*, 23, 1001–1018, 1994.

[21] U. Ferraro Petrillo, I. Finocchi, G. F. Italiano. The price of resiliency: a case study on sorting with memory faults. *Proc. 14th Annual European Symposium on Algorithms* (ESA'06), 768–779, 2006.

[22] I. Finocchi, F. Grandoni, and G. F. Italiano. Resilient search trees. *Proc. 18th ACM-SIAM Symposium on Discrete Algorithms* (SODA'07), 547–553, 2007.

[23] I. Finocchi, F. Grandoni, and G. F. Italiano. Optimal sorting and searching in the presence of memory faults. *Proc. 33rd Int. Colloquium on Automata, Lang. and Prog.* (ICALP'06), LNCS 4051 (part I), 286–298, 2006.

[24] I. Finocchi and G. F. Italiano. Sorting and searching in faulty memories. To appear in *Algorithmica*. Extended abstract in *Proc. 36th ACM Symposium on Theory of Computing* (STOC'04), 101–110, 2004.

[25] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. *Proc. IEEE Symposium on Security and Privacy*, 154–165, 2003.

[26] 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.

[27] M. R. Henzinger. The past, present and future of Web Search Engines. Invited talk. *31st Int. Coll. Automata, Languages and Programming* (ICALP'04), Turku, Finland, July 12–16 2004.

[28] M. R. Henzinger. Combinatorial Algorithms for Web Search Engines - Three Success Stories. Invited talk. *18th ACM-SIAM Symposium on Discrete Algorithms* (SODA'07), New Orleans, USA, January 7–9, 200.

[29] 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.

[30] A. G. Jorgensen, G. Moruz and T. Molhave. Priority queues resilient to memory faults. *Proc. 10th Workshop on Algorithms and Data Strutures* (WADS'07), 127–138, 2007.

[31] R. H. Katz, D. A. Patterson and G. A. Gibson, *Disk system architectures for high performance computing*, Proceedings of the IEEE, 77(12), 1842–1858, 1989.

[32] 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.

[33] K. B. Lakshmanan, B. Ravikumar, and K. Ganesan. Coping with erroneous information while sorting. *IEEE Trans. on Computers*, 40(9):1081–1084, 1991.

[34] 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.

[35] 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.

[36] T. C. May and M. H. Woods. Alpha-Particle-Induced Soft Errors In Dynamic Memories. *IEEE Trans. Elect. Dev.*, 26(2), 1979.

[37] 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.

[38] A. Pelc. Searching with known error probability. *Theoretical Computer Science*, 63, 185–202, 1989.

[39] A. Pelc. Searching games with errors: Fifty years of coping with liars. *Theoretical Computer Science*, 270, 71–109, 2002.

[40] B. Ravikumar. A fault-tolerant merge sorting algorithm. *Proc. 8th Annual Int. Conf. on Computing and Combinatorics* (COCOON'02), LNCS 2387, 440–447, 2002.

[41] A. Rényi. *A diary on information theory*, J. Wiley and Sons, 1994. Original publication: *Napló az információelméletről*, Gondolat, Budapest, 1976.

[42] S. Skorobogatov and R. Anderson. Optical fault induction attacks. *Proc. 4th International Workshop on Cryptographic Hardware and Embedded Systems* (CHES'02), LNCS 2523, 2–12, 2002.

[43] Tezzaron Semiconductor. *Soft errors in electronic memory - a white paper*, URL: `http://www.tezzaron.com/about/papers/Papers.htm`, January 2004.

[44] S. M. Ulam. *Adventures of a mathematician*. Scribners (New York), 1977.

[45] A.J. Van de Goor. *Testing semiconductor memories: Theory and practice*, ComTex Publishing, Gouda, The Netherlands, 1998.

[46] J. Von Neumann, Probabilistic logics and the synthesis of reliable organisms from unreliable components. In *Automata Studies*, C. Shannon and J. McCarty eds., Princeton University Press, 43–98, 1956.

[47] J. Xu, S. Chen, Z. Kalbarczyk, and R. K. Iyer. An experimental study of security vulnerabilities caused by errors. *Proc. International Conference on Dependable Systems and Networks*, 421–430, 2001.

[48] A. C. Yao and F. F. Yao. On fault-tolerant networks for sorting. *SIAM Journal on Computing*, 14, 120–128, 1985.