Deterministic History-Independent Strategies for Storing Information on Write-Once Memories*

Tal Moran† Moni Naor†‡ Gil Segev†

Abstract

Motivated by the challenging task of designing "secure" vote storage mechanisms, we deal with information storage mechanisms that operate in extremely hostile environments. In such environments, the majority of existing techniques for information storage and for security are susceptible to powerful adversarial attacks. In this setting, we propose a mechanism for storing a set of at most K elements from a large universe of size N on write-once memories in a manner that does not reveal the insertion order of the elements. We consider a standard model for write-once memories, in which the memory is initialized to the all 0's state, and the only operation allowed is flipping bits from 0 to 1. Whereas previously known constructions were either inefficient (required $\Theta(K^2)$ memory), randomized, or employed cryptographic techniques which are unlikely to be available in hostile environments, we eliminate each of these undesirable properties. The total amount of memory used by the mechanism is linear in the number of stored elements and poly-logarithmic in the size of the universe of elements.

In addition, we consider one of the classical distributed computing problems: conflict resolution in multiple-access channels. By establishing a tight connection with the basic building block of our mechanism, we construct the first deterministic and non-adaptive conflict resolution algorithm whose running time is optimal up to poly-logarithmic factors.

*A preliminary version of this work appeared as [16].

†Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot 76100, Israel. Email: {tal.moran,moni.naor,gil.segev}@weizmann.ac.il. Research supported in part by a grant from the Israel Science Foundation.

‡Incumbent of the Judith Kleeman Professorial Chair.

1 Introduction

We consider the abstract problem of storing a set of at most K elements taken from a large universe of size N, while minimizing the total amount of allocated memory. We design a storage mechanism which is deterministic, history-independent, and tamper-evident. Our mechanism supports insert operations, membership queries, and enumeration of all stored elements. Whereas previously known constructions were either inefficient, randomized, or employed cryptographic techniques that require secure key storage, we make a concentrated effort to eliminate these undesirable properties.

Our motivation emerges from the task of designing vote storage mechanisms[1], recently studied by Molnar, Kohno, Sastry and Wagner [15]. They described the desirable security goals of such mechanisms and suggested simple constructions. Without a "secure" vote storage mechanism, an adversary may be able to undetectably tamper with the voting records or compromise voter privacy. A typical threat is a corrupt poll worker who has complete access to the vote storage mechanism at some point during or after the election process. In order to prevent the adversary from modifying the stored votes, a vote storage mechanism should be *tamper-evident*. Moreover, as the order in which votes are cast must be hidden to protect the privacy of the voters, a vote storage mechanism should be *history-independent* as well. We refer the reader to [15] for a detailed list of the security goals of vote storage mechanisms.

In this paper we deal with the design of information storage mechanisms that operate in extremely hostile environments. In such environments, the majority of existing techniques for information storage and for security are susceptible to powerful adversarial attacks. In order to prevent such attacks, we study storage mechanisms with special properties.

Deterministic strategies. Randomization is an important ingredient in the design of efficient systems. However, for systems that operate in hostile environments, randomization can assist the adversary in attacking the system. First, as sources of random bits are typically obtained from the environment, it is quite possible that the adversary can corrupt these sources. In such cases, we usually have no guarantees on the expected behavior of the system. Second, even when truly random bits are available, these bits may be revealed to the adversary in advance, and serve as a crucial tool in the attack. Third, a randomized storage strategy may enable a subliminal channel: As multiple valid representations for the same abstract state exist, a malicious storage mechanism can secretly embed information into the stored data by choosing one of these representations. Applications such as voting protocols may run in completely untrusted environments. In such cases, deterministic strategies have invaluable security benefits.

History-independence. Many systems give away much more information than they were intended to. When designing a data structure whose memory representation may be revealed, we would like to ensure that an adversary will not be able to infer information that is not available through the system's legitimate interface. Computer science is rich with tales of cases where this was not done, such as files containing information whose creators assumed had been erased, only to be revealed later in embarrassing circumstances. Very informally, we consider a period of activity after which an adversary gains complete control over the data structure. In particular, the memory representation of the data is revealed to the adversary. The data structure is history-independent

[1] We note that for vote storage mechanisms it is sufficient to support only insert operations and enumeration of all stored elements. Our mechanism supports (efficient) membership queries as well, although we did not set out to implement this property.

if the adversary will not be able to deduce any more about the sequence of operations that led to the current content than the content itself yields (concrete definitions will be given in Section 1.3).

Tamper-evident write-once storage. A data structure is tamper-evident if any unauthorized modification of its content can be detected. Tamper-evidence is usually provided by a mixture of physical assumptions (such as secure processors) and cryptographic tools (such as signature schemes). Unfortunately, the majority of cryptographic tools require secure key storage, which is unlikely to be available in a hostile environment. Our construction follows the approach of Molnar et al. [15], who exploited the properties of write-once memories to provide tamper-evident storage. They introduced an encoding scheme in which flipping some of the bits of any valid codeword from 0 to 1 will never lead to another valid codeword[2]. In the voting scenario, this prevents any modification to the stored ballots after the polls close, and prevents poll workers from tampering with the content of the data structure while the storage device is in transit. This approach does not require any cryptographic tools or computational assumptions, which makes it very suitable for the setting of hostile environments. The additional memory allocation required by the encoding is only logarithmic in the size of the stored data, and can be handled independently of the storage strategy. For simplicity of presentation, we ignore the encoding procedure, and refer the reader's attention to the fact that our storage strategy is indeed write-once (i.e., the memory is initialized to the all 0's state, and the only operation allowed is flipping bits from 0 to 1).

Conflict resolution. In this paper we also address a seemingly unrelated problem: *conflict resolution in multiple-access channels*. A fundamental problem of distributed computing is to resolve conflicts that arise when several stations transmit simultaneously over a single channel. A conflict resolution algorithm schedules retransmissions, such that each of the conflicting stations eventually transmits singly to the channel. Such an algorithm is *non-adaptive* if the choice of the transmitting stations in each step does not depend on information gathered from previous steps. The efficiency measure for conflict resolution algorithms is the total number of steps it takes to resolve conflicts in the worst case[3].

We consider the standard model in which N stations are tapped into a single channel, and there are at most K conflicting stations. In 1985, Komlós and Greenberg [13] provided a *non-constructive* proof for the existence of a deterministic and non-adaptive algorithm that resolves conflicts in $O(K \log(N/K))$ steps. However, no explicit algorithm with a similar performance guarantee was known. By establishing a tight connection between this problem and the basic building block of our storage mechanism, we construct the first efficient deterministic and non-adaptive conflict resolution algorithm.

1.1 Our Contributions

We construct a deterministic mechanism for storing a set of at most K elements on write-once memories. The elements are given one at a time, and stored in a manner that does not reveal the insertion order. Our mechanism is immune to a large class of attacks that made previous constructions unsuitable for extremely hostile environments. Whereas previous constructions were either inefficient (required $\Theta(K^2)$ memory), randomized, or employed cryptographic techniques

[2]Consider the encoding $E(x) = x \;||\; \mathrm{wt}(\bar{x})_2$, obtained by concatenating the string x with the binary representation of the Hamming weight of its complement. Flipping any bit of x from 0 to 1 decreases $\mathrm{wt}(\bar{x})_2$, and requires flipping at least one bit of $\mathrm{wt}(\bar{x})_2$ from 1 to 0.

[3]Worst case refers to the maximum over all possible sets of conflicting stations.

that require secure key storage, we eliminate each of these undesirable properties. Our main result is the following[4]:

Theorem 1.1. *There exists an* explicit*, deterministic, history-independent, and write-once mechanism for storing a set of at most K elements from a universe of size N, such that:*

1. *The total amount of allocated memory is $O(K \cdot \text{polylog}(N))$.*
2. *The amortized insertion time and the worst-case look-up time are $O(\text{polylog}(N))$.*

In addition, our construction yields a non-constructive proof for the existence of the following storage mechanism:

Theorem 1.2. *There exists a deterministic, history-independent, and write-once mechanism for storing a set of at most K elements from a universe of size N, such that:*

1. *The total amount of allocated memory is $O(K \log(N/K))$.*
2. *The amortized insertion time is $O(\log(N/K))$.*
3. *The worst-case look-up time is $O(\log N \cdot \log K)$.*

Finally, by adapting our technique to the setting of conflict resolution, we devise the first efficient deterministic and non-adaptive algorithm for this problem. The number of steps required by our algorithm to resolve conflicts matches the non-explicit upper bound of Komlós and Greenberg [13] up to poly-logarithmic factors. More specifically, we prove the following theorem:

Theorem 1.3. *For every N and K there exists an explicit, deterministic, and non-adaptive algorithm that resolves any K conflicts among N stations in $O(K \cdot \text{polylog}(N))$ steps.*

Paper organization. The rest of the paper is organized as follows. In the remainder of this section we review related work and provide essential definitions. In Section 2 we present our main security goals and threat model. In Section 3 we present an overview of our storage mechanism, which is then described in Section 4. In Section 5 we provide constructions of the random-looking bipartite graphs that serve as the main building block of our storage mechanism. Finally, in Section 6 we show that our technique can be adapted to devise a deterministic and non-adaptive conflict resolution algorithm.

1.2 Related Work

The problem of constructing history-independent data structures was first formally considered by Micciancio [14], who devised a variant of 2–3 trees that satisfies a property of this nature. Micciancio considered a rather weak notion of history-independence, which required only that the *shape* of the trees does not leak information. We follow Naor and Teague [17] and consider a stronger notion – data structures whose *memory representation* does not leak information (see Section 1.3 for a formal definition and for related work that considered this definition and its variants). Naor and Teague focused on dictionaries, and constructed very efficient hash tables in which the cost of each operation is constant.

[4]For simplicity, throughout the paper we refer to the amount of allocated memory as the number of allocated memory *words*, each of length $\log N$ bits. We assume that the mechanism can read and write a memory word in constant time.

In the context of write-once memories, Rivest and Shamir [19] initiated the study of codes for write-once memory, by demonstrating that such memories can be "rewritten" to a surprising degree. Irani, Naor and Rubinfeld [11] explored the time and space complexity of computation using write-once memories, i.e., whether "a pen is much worse than a pencil". Specifically, they proved that a Turing machine with write-once polynomial space decides exactly the class of languages P.

History-independence on write-once memories. Molnar et al. [15] studied the task of designing a vote storage mechanism, and suggested constructions of history-independent storage mechanisms on write-once memories. Among their suggestions is a deterministic mechanism based on an observation of Naor and Teague [17], stating that one possible way of ensuring that the memory representation is determined by the content of a data structure is to store the elements in lexicographical order. This way, any set of elements has a single canonical representation, regardless of the insertion order of its elements. When dealing with write-once media, however, we cannot sort in-place when a new element is inserted. Instead, on every insertion, we compute the sorted list that includes the new element, copy the contents of this list to the next available memory position, and erase the previous list. We refer to this solution as a *copy-over list*, as suggested by Molnar et al. [15]. The main disadvantage of copy-over lists is that any insertion requires copying the entire list. Therefore, storing K elements requires $\Theta(K^2)$ memory[5].

In an attempt to improve the amount of allocated memory, Molnar et al. suggested using a hash table in which each entry is stored as a separate copy-over list. The copy-over lists are necessary when several elements are mapped to the same entry. However, with a fixed hash function the worst-case behavior of the table is very poor, and therefore the hash function must be randomly chosen and hidden from the adversary. Given the hash function, the mechanism is deterministic and we refer to such a strategy as an *off-line* randomized strategy. For instance, the mechanism may choose a pseudo-random function as its hash function. However, this approach is not suitable for hostile environments, where secure storage for the key of the hash function is not available.

Molnar et al. also showed that an *on-line* randomized strategy can significantly improve the amount of allocated memory. A simple solution is to allocate an array of $2K$ entries, and insert an element by randomly probing the array until an empty entry is found. However, as mentioned earlier, such a strategy may enable subliminal channels: a malicious storage mechanism can secretly embed information into the stored data by choosing among the multiple valid representations of the same data.

Tamper-evidence without write-once memories. Whereas the constructions of Molnar et al. achieved tamper-evidence by exploiting the properties of write-once memories, a different approach was taken by Bethencourt, Boneh and Waters [2]. They designed a history-independent tamper-evident storage mechanism by constructing a signature scheme which signs sets of elements such that the order in which elements were added cannot be determined, and elements cannot be deleted from the set. Even though their solution uses only $O(K)$ memory to store K elements, it is randomized and requires secure storage for cryptographic keys (as well as computational assumptions).

[5] When dealing with a small universe, a better solution is to pre-allocate memory to store a bounded unary counter for each element. However, this may not be suitable for vote storage in cases where write-in candidates are allowed (as common in the U.S.) or when votes are subsets or rankings (as common in many countries).

1.3 Formal Definitions

A data structure is defined by a list of operations. We construct a data structure that supports the following operations:

1. `Insert`(x) - stores the element x.
2. `Seal`() - finalizes the data structure (after this operation no `Insert` operations are allowed).
3. `LookUp`(x) - outputs `FOUND` if and only if x has already been stored.
4. `RetrieveAll`() - outputs all stored elements.

We say that two sequences of operations, S_1 and S_2, yield the same content if for all suffixes T, the results returned by T when the prefix is S_1 are identical to those returned by T when the prefix is S_2.

Definition 1.4. A deterministic data structure is *history-independent* if any two sequences of operations that yield the same content induce the same memory representation.

In our scenario, two sequences of operations yield the same content if and only if the corresponding sets of stored elements are identical. The above definition is a simplification of the one suggested by Naor and Teague [17], when dealing only with deterministic data structures. Naor and Teague also considered a stronger definition, in which the adversary gains control *periodically*, and obtains the current memory representation at several points along the sequence of operations. This definition has also been studied by Hartline et al. [10] and by Buchbinder and Petrank [3]. Since we deal only with deterministic data structures, in our setting the definitions are equivalent.

2 Security Goals and Threat Model

Our approach in defining the security goals and threat model is motivated by the possible attacks on an electronic voting system. To make the discussion clearer, we frame the threat model in terms of a vote storage mechanism. In an actual voting scenario, we think of ballot casting as an `Insert` operation. In the most trivial case, the element inserted is simply the chosen candidate's name. In more complex voting schemes, the inserted element may be a ranking of the candidates, an encrypted form of the ballot, or a combination of multiple choices. These possibilities are the reason for viewing the "universe of elements" as large, while the actual number of elements inserted is small (at most the number of voters). Once the voting is complete (e.g., the polls close), the `Seal` operation is performed. The purpose is to safeguard the ballots during transport (and for possible auditing). Finally, to count the votes, the `RetrieveAll` operation is performed. Note that in a standard voting scenario, `Lookup` is not needed – we provide it here only for completeness.

The main security goals we would like our storage mechanism to achieve are the following[6]:

1. **Tamper-evidence**: Any modification or deletion of votes after they were cast must be detected.
2. **Privacy**: No information about the order in which votes were cast should be revealed.
3. **Robustness**: No adversary should be able to cause the election process to fail.

[6]For simplicity we focus on the main and most relevant security goals. We refer the reader to [15] for a more detailed list.

We consider extremely powerful adversaries: Computationally unbounded adversaries that can adaptively corrupt any number of voters (i.e., the adversary can choose to perform arbitrary `Insert` operations at arbitrary points in time). In addition, we assume the adversary can modify the mechanism's program code *before* the election process begins, as long as the resulting code creates a "correct-looking" data structure (so that for any sequence of operations, a tester cannot distinguish between the memory representation output by the adversary's program and the memory representation output by a correct program). In our case, since our mechanism is deterministic, "correct-looking" is equivalent to actually being correct (whereas a randomized mechanism can enable in such a case a subliminal channel, as mentioned earlier).

The extent to which each of the above goals can be achieved depends on the assumed access that the adversary has to the mechanism. More specifically, we consider differing levels of adversarial access:

The full-access adversary. In the worst case, the adversary gains *complete* control of the mechanism at some point in time. After this point, the adversary has full read-write access both to the stored data and to the program code. The only limitation on the adversary's capabilities is our physical write-once assumption: The adversary cannot flip bits from 1 to 0 in the stored data. Such an adversary can always erase existing information (by setting all bits to 1), thus causing the election to fail. Such an adversary can also change the program code, so it can both record and change any votes cast. The best we can hope for in this case is a guarantee about operations that took place before the attack. The adversary should not be able to *undetectably* modify or delete votes that were previously cast, or to gain information about the order in which these votes were cast (beyond the order of the votes cast by corrupt voters).

The limited-access adversary. A slightly more optimistic case is when the adversary has write access only to the stored data, but not to the program code. Such an assumption may be plausible if the program code is stored in a read-only memory, or if the adversary gains access to the machine only after the election process has concluded. In this case, the adversary can still cause the election to fail (e.g., by erasing all stored data). However, the adversary should not be able to undetectably modify or delete any vote cast (except by corrupt voters). Moreover, the adversary should not gain any information about the order in which votes were cast beyond what would be gained by executing `RetrieveAll`() at any point in which the adversary has access to the mechanism.

The read-only adversary. In a best case scenario, the adversary has read-only access to the mechanism. This may occur if the adversary's access to the machine is through a programming bug, for instance. In this case, in addition to the security guarantees that we expect from the limited-access adversary, we would like to guarantee robustness as well: The adversary should not be able to cause the election to fail. This guarantee is not trivial, since the read-only adversary can still adaptively corrupt any number of voters.

Maliciously adding votes. An adversary with write access to the storage memory can always add votes (by simply executing the `Insert` operation). To detect such tampering, we combine two strategies: The first is adding a `Seal` operation to the mechanism, after which the `Insert` operation is no longer allowed. This will detect any tampering that occurs after the election process has concluded (as suggested by Molnar et al. [15]). The second is maintaining an independent count of the number of votes cast (e.g., voter lists maintained by election officials). Since our construction

guarantees that votes cannot be deleted, if an adversary adds bogus votes then the counts will not match up. Note that an independent count prevents adversaries from adding votes even if they gain access to the mechanism *during* the election process.

3 Overview of the Construction

Our construction relies on the fundamental technique of storing elements in a hash table and resolving collisions separately in each entry of the table. More specifically, our storage mechanism incorporates two "strategies": a *global strategy* that maps elements to the entries of the table, and a *local strategy* that resolves collisions that occur when several elements are mapped to the same entry. As long as both strategies are deterministic, history-independent and write-once, the entire storage mechanism will also share these properties.

The local strategy. We resolve collisions by storing the elements mapped to each entry of the table in a separate copy-over list. Copy-over lists were introduced by Molnar et al. [15], and are based on an observation by Naor and Teague [17], stating that one possible way of ensuring that the memory representation is determined by the content of a data structure is to store the elements in lexicographical order. When dealing with write-once media, however, we cannot sort in-place when a new element is inserted. Instead, on every insertion, we compute the sorted list that includes the new element, copy the contents of this list to the next available memory position, and erase the previous list (by setting all the bits to 1). Note that storing K elements in a copy-over list requires $\Theta(K^2)$ memory, and therefore is reasonable only for small values of K.

The global strategy. Our goal is to establish a *deterministic* strategy for mapping elements to the entries of the table. However, for any fixed hash function, the set of inserted elements can be chosen such that the load in at least one of the entries will be too high to be efficiently handled by our local strategy. Therefore, in order to ensure that the number of elements mapped to each entry remains relatively small (in the worst case), we must apply a more sophisticated strategy.

Our global strategy stores the elements in a *sequence* of tables, where each table enables us to store a fraction of the elements. Each element is first inserted into *several* entries of the first table. When an entry overflows (i.e., more than some pre-determined number of elements are inserted into it), the entry is "permanently deleted". In this case, any elements that were stored in this entry and are not stored elsewhere in the table are inserted into the next table in a similar manner. Thus, we are interested in finding a sequence of functions that map the universe of elements to the entries of the tables, such that the total number of tables, the size of each table, and the number of collisions are minimized. We view such functions as bipartite graphs $G = (L, R, E)$, where the set of vertices on the left, L, is identified with the universe of elements, and the vertices on the right, R, are identified with the entries of a table. Given a set of elements $S \subseteq L$ to store, the number of elements mapped to each table entry $y \in R$ is the number of neighbors that y has from the set S. We would like the set $S \subseteq L$ to have as few as possible overflowing entries, i.e., as few as possible vertices $y \in R$ with many neighbors in S.

More specifically, we are interested in bipartite graphs $G = (L, R, E)$ with the following property: Every set $S \subseteq L$ of size at most K contains "many" vertices with low-degree neighbors. We refer to such graphs as *bounded-neighbor expanders*[7]. Our global strategy will map all the elements

[7] The definition is motivated by the notion of bipartite unique-neighbor expanders presented by Alon and Capalbo [1].

in S which have a low-degree neighbor to those neighbors, and this guarantees that the table entries corresponding to those neighbors will not overflow at any stage. However, not every element in S will have a low-degree neighbor. For this reason, we use a sequence of bipartite graphs, all sharing the same left set L. Each graph will enable us to store a fraction of the elements in S. Formally, we define:

Definition 3.1. Let $G = (L, R, E)$ be a bipartite graph. We say that a vertex $x \in L$ has an *ℓ-degree neighbor with respect to $S \subseteq L$*, if it has a neighbor $y \in R$ with no more than ℓ incoming edges from S.

Definition 3.2. A bipartite graph $G = (L, R, E)$ is a *(K, α, ℓ)-bounded-neighbor expander*, if every $S \subseteq L$ of size K contains at least $\alpha|S|$ vertices that have an ℓ-degree neighbor with respect to S.

We denote $|L| = N$. In addition, we assume that all the vertices on the left side have the same degree D. We discuss and provide constructions of bounded-neighbor expander graphs in Section 5.

4 The Construction

Let $G_0, \ldots, G_t$ denote a sequence of bounded-neighbor expanders $G_i = (L = [N], R_i, E_i)$ with left-degree D_i. The graphs are constructed such that:

- G_0 is a $(K_0 = K, \alpha_0, \ell_0)$-bounded-neighbor expander, for some α_0 and ℓ_0.
- For every $1 \leq i \leq t$, G_i is a (K_i, α_i, ℓ_i)-bounded-neighbor expander, for some α_i and ℓ_i, where $K_i = (1 - \alpha_{i-1})K_{i-1}$.

As described in Section 3, the elements are stored in a sequence of tables, $T_0, \ldots, T_t$. Each table T_i is identified with the right set R_i of the bipartite graph G_i, and contains $|R_i|$ entries denoted by $T_i[1], \ldots, T_i[|R_i|]$. The elements are mapped to the entries of the tables and are stored there using a separate copy-over list at each entry. The copy-over list at each entry of table T_i will store at most ℓ_i elements. We denote by $|T_i[y]|$ the number of elements stored in the copy-over list $T_i[y]$, and use the notation $T_i[y] = *$ to indicate that the copy-over list $T_i[y]$ overflowed and was permanently deleted.

In order to insert or look-up an element x, we execute $\texttt{Insert}(x, T_0)$ or $\texttt{LookUp}(x, T_0)$, respectively. The $\texttt{Seal}()$ operation is performed as in [15] by using the encoding discussed in the introduction. The operations $\texttt{Insert}(x, T_i)$, $\texttt{LookUp}(x, T_i)$, and $\texttt{RetrieveAll}()$ are described in Figure 1.

We first prove that the storage mechanism is history-independent, i.e., any two sequences of insertions that yield the same content, induce the same memory representation. Then, we show that each table indeed stores a fraction of the elements. Finally we summarize the properties of the constructions.

Lemma 4.1. *For every set $S \subseteq [N]$ of size at most K, any insertion order of its elements induces the same memory representation.*

Proof. Let $S \subseteq [N]$ of size at most K. We prove by induction on $0 \leq i \leq t$ that the memory representation of table T_i is independent of the insertion order.

```
Insert(x, T_i):
 1: for all neighbors y of x in the graph G_i do
 2:    if T_i[y] = * then
 3:       Continue to the next neighbor of x
 4:    else if |T_i[y]| < ℓ_i then
 5:       Store x in the copy-over list T_i[y]
 6:    else
 7:       for all x' in T_i[y] such that x' does not appear in any other list in T_i do
 8:          Execute Insert(x', T_{i+1})
 9:       Set T_i[y] ← *  // erase the memory blocks of T_i[y]
10: if x was not stored in any copy-over list in the previous step then
11:    Execute Insert(x, T_{i+1})

LookUp(x, T_i):
 1: for all neighbors y of x in the graph G_i do
 2:    if x is stored in the copy-over list T_i[y] then
 3:       return FOUND and halt
 4: if x was not found in a previous step and i = t then
 5:    return NOT FOUND
 6: else
 7:    return LookUp(x, T_{i+1})

RetrieveAll():
 1: for all tables T_i do
 2:    for all copy-over lists T_i[y] do
 3:       if T_i[y] ≠ * then
 4:          Output all elements of T_i[y] that have not yet been output
```

Figure 1: The `Insert`, `LookUp`, and `RetrieveAll` operations.

For $i = 0$, denote by Y_0 the set of vertices in R_0 that have no more than ℓ_0 incoming edges from S in the graph G_0. Then, it is clear that for every $y \in Y_0$ and for any insertion order, the copy-over list in entry $T_0[y]$ never contains more than ℓ_0 elements, and therefore never erased. Moreover, this list will always contain the same elements (all the neighbors of y in S) which will be stored in a history-independent manner. In addition, for every $y \in R_0 \setminus Y_0$ and for any insertion order, the copy-over list at entry $T_0[y]$ will be erased at some point (since it will exceed the ℓ_0 upper bound), and will contain a fixed number of erased blocks. Therefore, the memory representation of T_0 is independent of the insertion order.

Suppose now that the memory representation of $T_0, \ldots, T_{i-1}$ is independent of the insertion order. In particular this implies that for every set S there exists a fixed $S_i \subset S$ such that the elements of S_i are all stored in $T_0, \ldots, T_{i-1}$. Let $S'_i = S \setminus S_i$. Then, in any insertion order, only the elements of S'_i are inserted into table T_i (not necessarily successfully, i.e., will remain in T_i). Now, denote by Y_i the set of vertices in R_i that have no more than ℓ_i incoming edges from S_i in the graph G_i. Then, for every $y \in Y_i$ and for any insertion order, the copy-over list in entry $T_i[y]$ will never contain more than ℓ_i elements, and therefore will store all the neighbors of y from S'_i in a history independent manner. In addition, for every $y \in R_i \setminus Y_i$ and for any insertion order, the copy-over list at entry $T_i[y]$ will be erased at some point (since it will exceed the ℓ_i upper bound), and will contain a fixed number of erased blocks. Therefore, the memory representation of T_i is independent of the insertion order as well. ■

Lemma 4.2. *For every set $S \subseteq [N]$ of size at most K, for every insertion order of its elements, and for every $0 \leq i \leq t$, the number of* `Insert`$(\cdot, T_i)$ *calls is at most K_i. In particular, if there exists an $\alpha > 0$ such that $\alpha_i \geq \alpha$ for every G_i, then setting $t = \left\lceil \frac{\ln K}{\alpha} \right\rceil$ guarantees that every such set S is successfully stored.*

Proof. We prove the first part of the lemma by induction on i. For $i = 0$, it is clear that the number of `Insert`$(\cdot, T_0)$ calls is at most $K_0 = K$, since S contains at most K elements.

Suppose now that the number of `Insert`$(\cdot, T_i)$ calls is at most K_i. Fix an insertion ordering, and denote by S_i the set of elements x for which an `Insert`(x, T_i) call was executed. An element x' will be inserted by `Insert`(x', T_{i+1}) only if it was previously inserted by `Insert`(x', T_i), and then either did not find an available copy-over list to enter, or was erased when a copy-over list exceeded the ℓ_i upper bound. Notice that in the graph G_i, if some $x \in S_i$ has a neighbor $y \in R_i$ with at most ℓ_i incoming edges from S_i, then x will be successfully placed in the copy-over list $T_i[y]$. This is due to the fact that y has at most ℓ_i incoming edges from S_i, and therefore the copy-over list $T_i[y]$ will not be erased.

This implies that the number of `Insert`$(\cdot, T_{i+1})$ calls is upper bounded by the number of vertices in S_i which do not have an ℓ_i-degree neighbor with respect to S_i in G_i. We now claim that the number of such vertices is at most $(1 - \alpha_i)K_i = K_{i+1}$. Extend S_i arbitrarily to a set S'_i of size *exactly* K_i. Then, Definition 3.2 implies there are at least αK_i vertices in S'_i that have an ℓ_i-degree neighbor with respect to S'_i. Since $S_i \subseteq S'_i$, then any vertex that has an ℓ_i-degree neighbor with respect to S'_i, also has (the same) ℓ_i-degree neighbor with respect to S_i. Therefore, there are at least $\alpha_i K_i$ vertices in S'_i that have an ℓ_i-degree neighbor with respect to S_i. This implies that at most $(1 - \alpha_i)K_i$ vertices in S'_i do not have an ℓ_i-degree neighbor with respect to S_i. In particular, since $S_i \subseteq S'_i$, there are at most $(1 - \alpha_i)K_i$ vertices in S_i that do not have an ℓ_i-degree neighbor with respect to S_i.

Now, if there exists an $\alpha > 0$ such that $\alpha_i \geq \alpha$ for every G_i, then in particular the number of `Insert`$(\cdot, T_t)$ calls is at most

$$K_t = K \cdot \prod_{i=0}^{t-1}(1 - \alpha_i) \leq K \cdot (1 - \alpha)^t \leq K \cdot e^{-\alpha t} \leq 1 \ .$$

Thus, at most one element is inserted into the last table T_t, and therefore the set S is successfully stored in the sequence of $t + 1$ tables. ∎

Lemma 4.3. *The storage mechanism has the following properties:*

1. *The total amount of allocated memory is at most $\sum_{i=0}^{t} |R_i| \cdot \ell_i^2$.*
2. *The* amortized *insertion time is at most $\frac{1}{K} \cdot \left(\sum_{i=0}^{t} |R_i| \cdot \ell_i^2\right)$.*
3. *The* worst-case *look-up time is at most $\sum_{i=0}^{t} D_i \cdot \ell_i^2$.*

Proof. Each table T_i contains $|R_i|$ entries, each of which stores at most ℓ_i elements in a copy-over list by using at most ℓ_i^2 memory blocks. Therefore, the total amount of allocated memory is at most $\sum_{i=0}^{t} |R_i| \cdot \ell_i^2$. Moreover, since the storage strategy is write-once, then the total time for storing K elements is upper bounded by the amount of memory in which the elements are stored. Therefore, the amortized insertion time is at most $\frac{1}{K} \cdot \left(\sum_{i=0}^{t} |R_i| \cdot \ell_i^2\right)$. When searching for an element, each table has to be accessed at only D_i entries, where each entry contains at most ℓ_i^2 memory blocks. Therefore, The worst-case look-up time is at most $\sum_{i=0}^{t} D_i \cdot \ell_i^2$. ∎

Theorems 1.1 and 1.2 now follow by instantiating the mechanism with the bounded-neighbor expanders from Corollary 5.7 and Theorem 5.1, respectively. The proof of Theorem 1.1 is provided below, and the proof of Theorem 1.2 is almost identical and therefore omitted.

Proof of Theorem 1.1. When the sequence of graphs $G_0, \ldots, G_t$ are constructed according to Corollary 5.7 with $\epsilon = 1/2$, we have that every G_i is a (K_i, α_i, ℓ_i)-bounded-neighbor expander, such that

$$\alpha_i = \frac{|R_i|}{4D_iK_i} \geq \frac{cK_i}{\log^3(N)} \cdot \frac{1}{4D_iK_i} = \frac{c}{4D_i \log^3(N)} = \frac{c}{4D \log^3(N)} \ ,$$

for some constant $c > 0$ and $D = \text{polylog}(N)$. Therefore, by Lemma 4.2, we can set $t = \left\lceil \frac{\ln K}{\alpha} \right\rceil$, where $\alpha = \frac{c}{4D \log^3(N)}$. Now, Lemma 4.3 states that the total amount of allocated memory is

$$\begin{aligned} \sum_{i=0}^{t} |R_i| \cdot \ell_i^2 &= \sum_{i=0}^{t} |R_i| \cdot \left(\frac{4D_iK_i}{|R_i|} \right)^2 = 16D^2 \sum_{i=0}^{t} \frac{K_i^2}{|R_i|} \leq \frac{16D^2 \log^3(N)}{c} \sum_{i=0}^{t} K_i \\ &\leq \frac{16KD^2 \log^3(N)}{c} \sum_{i=0}^{t} (1-\alpha)^i \leq \frac{16KD^2 \log^3(N)}{c} \cdot \frac{1}{\alpha} = \frac{64KD^3 \log^6(N)}{c^2} \ . \end{aligned}$$

Thus, the required memory allocation is $O(K \cdot \text{polylog}(N))$. Lemma 4.3 further implies that the amortized insertion time and the worst-case look-up time are $O(\text{polylog}(N))$. ∎

5 Constructions of Bounded-Neighbor Expanders

Given N and K we are interested in constructing a (K, α, ℓ)-bounded-neighbor expander $G = (L = [N], R, E)$, such that α is maximized, and ℓ and $|R|$ are minimized. We first present a non-constructive proof of the existence of a bounded-neighbor expander that enjoys "the best of the two worlds": $\alpha = 1/2$, $\ell = 1$, and almost linear $|R|$. Then, we provide an explicit construction of bounded-neighbor expanders, by showing that any disperser [20] is in fact a bounded-neighbor expander.

5.1 A Non-Constructive Proof

We prove the following theorem:

Theorem 5.1. *For every N and K, there exists a $(K, 1/2, 1)$-bounded-neighbor expander $G = (L, R, E)$, with $|L| = N$, $|R| = O(K \log(N/K))$ and left-degree $D = O(\log(N/K))$.*

In order to prove the theorem, we show that for every N and K, there exists a family $\mathcal{H}$ containing $O(\log(N/K))$ functions $h : [N] \to [3K]$ with the following property: For every $S \subseteq [N]$ of size K, there exists a function $h \in \mathcal{H}$ such that h restricted to S maps at least $K/2$ elements of S to unique elements of $[3K]$. Alternatively, we can view each function h as a bipartite graph $G_h = ([N], [3K], E_h)$, where $(x, y) \in E_h$ if and only if $h(x) = y$, and ask that for every $S \subseteq [N]$ of size K there exists a function $h \in \mathcal{H}$ such that at least $K/2$ elements in S have 1-degree neighbors with respect to S in G_h.

Given such a family $\mathcal{H} = \{h_1, \ldots, h_t\}$, we define a bipartite graph $G = (L = [N], R, E)$ where R contains $t = O(\log(N/K))$ copies of $[3K]$. Each copy represents a function in $\mathcal{H}$. More specifically, each vertex $x \in [N]$ has t outgoing edges, where the i-th edge is connected to $h_i(x)$ in the i-th copy of $[3K]$. See Figure 2 for an illustration of the constructed graph.

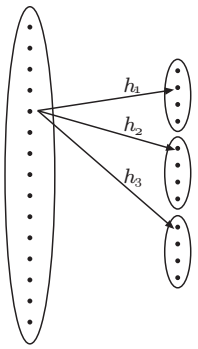

Figure 2: The constructed bounded-neighbor expander for the case $t = 3$.

Lemma 5.2. *Let X denote the number of bins that contain exactly one ball, when K balls are placed independently and uniformly at random in $3K$ bins. Then,*

$$\Pr[X < K/2] < \exp(-K/48) \ .$$

Proof. For every $1 \leq i \leq K$, denote by X_i the Boolean random variable that equals 1 if and only if the i-th ball is placed in a bin that does not contain any other balls. Then $X = \sum_{i=1}^{K} X_i$. Note that since K balls are placed in $3K$ bins, then there are always at least $2K$ empty bins. Therefore, for every $\vec{u} \in \{0,1\}^{K-1}$ and for every $1 \leq i \leq K$,

$$\Pr[X_i = 1 \mid (X_1, \ldots, X_{i-1}, X_{i+1}, \ldots, X_K) = \vec{u}] \geq 2/3 \ .$$

Let $Y_1, \ldots, Y_K$ denote K independent and identically distributed Boolean random variables such that $\Pr[Y_1 = 1] = 2/3$, and let $Y = \sum_{i=1}^{K} Y_i$. A standard coupling argument shows that for every $t > 0$ it holds that $\Pr[X < t] \leq \Pr[Y < t]$. Therefore, by applying a Chernoff bound for Y, we obtain

$$\Pr[X < K/2] \leq \Pr[Y < K/2] \leq \exp(-K/48) \ .$$

■

The following lemma proves the existence of the family $\mathcal{H}$, which is used to construct the bounded-neighbor expander as explained above.

Lemma 5.3. *For every N and K, there exists a family $\mathcal{H}$ containing $O(\log(N/K))$ functions $h : [N] \to [3K]$, such that for every $S \subseteq [N]$ of size K, there exists a function $h \in \mathcal{H}$ whose restriction to S maps at least $K/2$ elements of S to unique elements of $[3K]$.*

Proof. We apply a hitting-set argument. For every fixed $S \subseteq [N]$ of size K, Lemma 5.2 implies that with probability at least $1 - \exp(-K/48)$ over the choice of a random function $h : [N] \to [3K]$, we have that the restriction of h to S maps at least $K/2$ elements of S to unique elements of $[3K]$. Therefore, there exists a function h_1 which have this property with respect to at least $(1 - \exp(-K/48)) \binom{N}{K}$ sets of size K. We eliminate all of these sets, and start again with the remaining sets. This way, we get that the size of the family $\mathcal{H}$ satisfies

$$(\exp(-K/48))^{|\mathcal{H}|} \binom{N}{K} < 1 \ ,$$

which is obtained for $|\mathcal{H}| = O(\log(N/K))$. ■

5.2 An Explicit Construction

We provide an explicit construction of bounded-neighbor expanders by showing that any disperser is a bounded-neighbor expander. Dispersers [20] are combinatorial objects with many random-like properties. Dispersers can be viewed as functions that take two inputs: a string that is not uniformly distributed, but has some randomness; and a shorter string that is completely random, and output a string whose distribution is guaranteed to have a large support. Dispersers have found many applications in computer science, such as simulation with weak sources, deterministic amplification, and many more (see [18] for a comprehensive survey). We now formally define Dispersers, and then show that any disperser is a bounded-neighbor expander.

Definition 5.4. A bipartite graph $G = (L, R, E)$ is a (K, ϵ)-disperser if for every $S \subseteq L$ of size at least K, it holds that $|\Gamma(S)| \geq (1-\epsilon)|R|$, where $\Gamma(S)$ denotes the set of neighbors of the vertices in S.

Lemma 5.5. *Any (K, ϵ)-disperser $G = (L, R, E)$ with left-degree D is a (K, α, ℓ)-bounded-neighbor expander, for $\alpha = \frac{(1-\epsilon)|R|}{2DK}$ and $\ell = 1/\alpha$.*

Proof. We have to show that every set $S \subseteq L$ of size K contains least $\alpha|S|$ vertices that have an ℓ-degree neighbor with respect to S (that is, a neighbor that has at most ℓ incoming edges from S). We can focus only the subgraph $G' = (S, \Gamma(S), E')$, where E' are all the outgoing edges of S. There are exactly DK edges in G', and therefore the average degree of the vertices of $\Gamma(S)$ in G' is

$$\frac{DK}{|\Gamma(S)|} \leq \frac{DK}{(1-\epsilon)|R|} = \frac{\ell}{2} \ .$$

This implies that at least $|\Gamma(S)|/2$ vertices in $\Gamma(S)$ have degree at most ℓ in G'. Thus, the number of vertices in S which have an ℓ-degree neighbor with respect to S is at least

$$\frac{|\Gamma(S)|}{2D} \geq \frac{(1-\epsilon)|R|}{2D} = \frac{(1-\epsilon)|R|}{2DK} \cdot |S| = \alpha|S| \ .$$

■

Lemma 5.5 can be instantiated with the following disperser construction of Ta-Shma, Umans and Zuckerman [22].

Theorem 5.6 ([22])**.** *For every n, k, and constant $\epsilon > 0$, there exists an efficiently computable $(K = 2^k, \epsilon)$-disperser $G = (L, R, E)$, with $|L| = N = 2^n$, $|R| = \Theta(K/\log^3(N))$ and left-degree $D = \text{polylog}(N)$.*

Corollary 5.7. *For every n, k and constant $\epsilon > 0$, there exists an efficiently computable $(K = 2^k, \alpha, 1/\alpha)$-bounded-neighbor expander $G = (L, R, E)$, with $|L| = N = 2^n$, $|R| = \Theta(K/\log^3(N))$, left-degree $D = \text{polylog}(N)$, and $\alpha = \frac{(1-\epsilon)|R|}{2DK}$.*

An alternative approach for constructing bounded-neighbor expanders is via loss-less condensers[8]. This approach guarantees constant α and very small ℓ, but larger $|R|$. The recent construction of Guruswami, Umans and Vadhan [9] yields a bounded-neighbor expander with $|R| = O(K^{1+\epsilon})$, for every constant $\epsilon > 0$. Therefore, this is preferable only when dealing with relatively small values of K, such as $K = \text{polylog}(N)$.

[8] We note that unbalanced expanders have been already considered for storing sets of elements by Buhrman, Miltersen, Radhakrishnan and Venkatesh [4] and by Ta-Shma [21] with the property that membership queries can be answered by querying just one bit.

6 A Deterministic Non-Adaptive Conflict Resolution Algorithm

In the conflict resolution problem, N stations are tapped into a multiple-access channel, and the goal is to resolve conflicts that arise when K stations transmit simultaneously over the channel. A conflict resolution algorithm schedules retransmissions, such that each of the conflicting stations eventually transmits singly to the channel. At each step, if more than one station transmits, then all packets are lost. After each step the transmitting stations receive feedback indicating only the success or failure of their transmission. A station that successfully transmits halts, and waits for the algorithm to terminate.

A conflict resolution algorithm is *non-adaptive* if the choice of the transmitting stations in each step does not depend on information gathered from previous steps. The efficiency measure for conflict resolution algorithms is the total number of steps it takes to resolve conflicts in the worst case, where worst case refers to the maximum over all possible sets of K conflicting stations.

Several deterministic adaptive solutions are known. Capetanakis's tree algorithms [5, 6], that resolve conflicts in $O(K \log(N/K))$ steps, were devised almost three decades ago. Greenberg and Winograd [8] showed that any deterministic algorithm must run for $\Omega(K(\log N)/\log K)$ steps. In 1985, Komlós and Greenberg [13] provided a *non-constructive* proof for the existence of a deterministic and non-adaptive algorithm that resolves conflicts in $O(K \log(N/K))$ steps. However, no explicit algorithm with a similar performance guarantee was known. As noted by Komlós and Greenberg, a very simple deterministic and non-adaptive algorithm can resolve conflicts in $O(K^2 \log N)$ steps. This simple solution will be used by our algorithm in order to "locally" resolve a small number of conflicts.

6.1 Overview of the Algorithm

We adapt the main idea underlying our storage mechanism by following similar "strategies": A *global strategy* that maps stations to time intervals, and a *local strategy* that schedules retransmissions inside the intervals. The global strategy is identical to that of the storage mechanism: We map the N stations to time intervals using a sequence of bounded-neighbor expanders. The local strategy schedules retransmissions inside the intervals by associating the stations with codewords of a superimposed code [12]. Given N and ℓ, a binary superimposed code of size N guarantees that any codeword is not contained in the bit-wise or of any other $\ell - 1$ codewords. We use the following result of Erdös, Frankel and Füredi [7].

Theorem 6.1 ([7]). *For every N and ℓ there exists an efficiently computable code $C : [N] \to \{0,1\}^d$ where $d \leq 16\ell^2 \log N$, such that for every distinct $x_1, \ldots, x_\ell \in [N]$ it holds that $C(x_1) \subsetneq \bigvee_{i=2}^{\ell} C(x_i)$.*

In every interval, we associate each station x that is mapped to the interval with a codeword $C(x) \in \{0,1\}^d$. Each interval contains d steps, and the station x transmits at its j-th step if and only if the j-th entry of $C(x)$ is 1. The superimposed code guarantees that if at most ℓ stations are mapped to an interval, then each station will successfully transmit. This approach provides a deterministic and non-adaptive algorithm that resolves conflicts among any ℓ stations in $d = O(\ell^2 \log N)$ steps.

6.2 The Algorithm

Let $G_0, \ldots, G_t$ denote a sequence of bounded-neighbor expanders $G_i = (L = [N], R_i, E_i)$ with left-degree D_i, and let $C_0, \ldots, C_t$ denote a sequence of codes $C_i : [N] \to \{0,1\}^{d_i}$. The graphs and codes are constructed such that:

- G_0 is a $(K_0 = K, \alpha_0, \ell_0)$-bounded-neighbor expander, for some α_0 and ℓ_0.
- For every $1 \leq i \leq t$, G_i is a (K_i, α_i, ℓ_i)-bounded-neighbor expander, for some α_i and ℓ_i, where $K_i = (1 - \alpha_{i-1})K_{i-1}$.
- For every $0 \leq i \leq t$, C_i has the property that for every distinct $x_1, \ldots, x_{\ell_i} \in [N]$ it holds that $C_i(x_1) \subsetneq \bigvee_{j=2}^{\ell_i} C_i(x_j)$.

The algorithm runs in a sequence of intervals $I_0, \ldots, I_t$. Each interval T_i is identified with the right set R_i of the bipartite graph G_i, and is divided into $|R_i|$ sub-intervals denoted by $I_i[1], \ldots, I_i[|R_i|]$. A station $x \in [N]$ participates in sub-interval $I_i[y]$ if and only if x is adjacent to y in the graph G_i. The sub-interval $I_i[y]$ contains d_i steps, and a participating station x transmits at its j-th step if and only if the j-th entry of $C_i(x)$ is 1.

The following lemma summarizes the properties of the algorithm. The proof is almost identical to the proof of the corresponding properties of the storage mechanism in Section 4, and is therefore omitted. Theorem 1.3 is proved by instantiating the algorithm with the explicit family of bounded-neighbor expanders constructed in Section 5. Again, the proof is almost identical to the proof of Theorem 1.1, and is omitted.

Lemma 6.2. *The following properties hold:*

1. *For every set of K conflicting stations and for every $0 \leq i \leq t$, the number of active stations at the beginning of interval I_i is at most K_i.*
2. *For every set of K conflicting stations, the algorithm terminates in $16 \left(\sum_{i=0}^{t} |R_i| \cdot \ell_i^2\right) \cdot \log N$ steps.*

7 Concluding Remarks

Non-amortized insertion time. The amortized insertion time of our storage mechanism is at most poly-logarithmic. However, the worst-case insertion time may be larger, since an insertion may have a cascading effect. In some cases, this might enable a side-channel attack in which the adversary exploits the insertion times in order to obtain information on the order in which elements were inserted. We note that if multiple writes are allowed, then by combining our global strategy with the hashing method of Naor and Teague [17], we can achieve a poly-logarithmic worst-case insertion time, as well as linear memory allocation. Whether this is possible using write-once memory remains an open problem.

Bounded-neighbor expanders. The explicit construction of bounded-neighbor expanders in Section 5 does not achieve the parameters that one can hope for according to Theorem 5.1. It would be interesting to improve our explicit construction, as any such improvement will in turn lead to a more efficient instantiation of our storage mechanism.

Memory lower bound. The total amount of allocated *bits* required by the mechanism stated in Theorem 1.2 is $O(K \log(N) \log(N/K))$. This leaves a gap between the optimal construction using multiple-writes (that requires only $O(K \log(N/K))$ bits) and our construction using write-once memory. An interesting open question is whether this gap is an unavoidable consequence of using write-once memory. This can be alternatively formulated as follows: find the minimal integer

$M = M(N, K)$ such that any set $S \subseteq [N]$ of size at most K can be mapped to a set $V_S \subseteq [M]$, with the property that $V_{S_1} \subseteq V_{S_2}$ whenever $S_1 \subseteq S_2$. Note that any solution to this vector assignment problem can be translated into a write-once strategy that requires a memory of size M bits.

Dealing with multi-sets. Our storage mechanism can be easily adapted to store *multi-sets* of K elements taken from a universe of size N. This setting can be viewed as dealing with a universe of size $N' = NK$, and storing an element $x \in [N]$ as (x, i) where $i \in [K]$ is the appearance number of the element. Note that in order to insert an element x we first need to retrieved its current number of appearances. This number can be retrieved using $\log K$ invocations of the `LookUp` procedure in order to identify the maximal $i \in [K]$ such that (x, i) is stored (using a binary search). These modifications only add poly-logarithmic factors to the performance of the mechanism, and therefore Theorem 1.1 holds in this setting as well.

Robustness, efficiency and "lunchtime attacks". A limited-access adversary may launch a "lunchtime attack": an attack in which the adversary has a short window of write-access to the machine (but cannot change its program memory). In this case, depending on the error handling procedures for the algorithm, it may be possible for the history-independence property of the data structure to be violated. For example, the adversary can make sure a specific item cannot be inserted by erasing all the relevant memory locations. If such an error causes the machine to halt, then the adversary will be able to determine which items were inserted before the "marked" item. A trivial way to prevent this is by verifying that the data structure is correct before each `Insert` operation. This, however, increases the amortized time complexity of insertion considerably. By sacrificing some robustness properties of the data structure under such an attack (e.g., flagging errors, but possibly loosing data) it is possible to maintain history-independence without a loss of efficiency.

Acknowledgments

We thank Ronen Gradwohl for many useful discussions and suggestions.

References

[1] N. Alon and M. R. Capalbo. Explicit unique-neighbor expanders. In *Proceedings of the 43rd Symposium on Foundations of Computer Science*, pages 73–79, 2002.

[2] J. Bethencourt, D. Boneh, and B. Waters. Cryptographic methods for storing ballots on a voting machine. In *Proceedings of the 14th Network and Distributed System Security Symposium*, pages 209–222, 2007.

[3] N. Buchbinder and E. Petrank. Lower and upper bounds on obtaining history-independence. *Information and Computation*, 204(2):291–337, 2006. A preliminary version appeared in *Advances in Cryptology - CRYPTO '03*, pages 445–462, 2003.

[4] H. Buhrman, P. B. Miltersen, J. Radhakrishnan, and S. Venkatesh. Are bitvectors optimal? *SIAM Journal on Computing*, 31(6):1723–1744, 2002. A preliminary version appeared in *Proceedings of the 32nd Annual ACM Symposium on Theory of Computing*, pages 449–458, 2000.

[5] J. Capetanakis. Generalized TDMA: The multi-accessing tree protocol. *IEEE Transactions on Communications*, 27(10):1479–1484, 1979.

[6] J. Capetanakis. Tree algorithms for packet broadcast channels. *IEEE Transactions on Information Theory*, 25(5):505–515, 1979.

[7] P. Erdös, P. Frankel, and Z. Füredi. Families of finite sets in which no set is covered by the union of r others. *Israeli Journal of Mathematics*, 51:79–89, 1985.

[8] A. G. Greenberg and S. Winograd. A lower bound on the time needed in the worst case to resolve conflicts deterministically in multiple access channels. *Journal of the ACM*, 32(3):589–596, 1985.

[9] V. Guruswami, C. Umans, and S. Vadhan. Unbalanced expanders and randomness extractors from parvaresh-vardy codes. In *Proceedings of the 22nd Annual IEEE Conference on Computational Complexity*, to appear, 2007.

[10] J. D. Hartline, E. S. Hong, A. E. Mohr, W. R. Pentney, and E. Rocke. Characterizing history independent data structures. *Algorithmica*, 42(1):57–74, 2005.

[11] S. Irani, M. Naor, and R. Rubinfeld. On the time and space complexity of computation using write-once memory -or- Is pen really much worse than pencil? *Mathematical Systems Theory*, 25(2):141–159, 1992.

[12] W. H. Kautz and R. C. Singleton. Nonrandom binary superimposed codes. *IEEE Transactions on Information Theory*, 10(3):363–377, 1964.

[13] J. Komlós and A. G. Greenberg. An asymptotically fast nonadaptive algorithm for conflict resolution in multiple-access channels. *IEEE Transactions on Information Theory*, 31(2):302–306, 1985.

[14] D. Micciancio. Oblivious data structures: Applications to cryptography. In *Proceedings of the 29th Annual ACM Symposium on the Theory of Computing*, pages 456–464, 1997.

[15] D. Molnar, T. Kohno, N. Sastry, and D. Wagner. Tamper-evident, history-independent, subliminal-free data structures on PROM storage -or- How to store ballots on a voting machine (extended abstract). In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 365–370, 2006. The full version is available from the Cryptology ePrint Archive, Report 2006/081.

[16] T. Moran, M. Naor, and G. Segev. Deterministic history-independent strategies for storing information on write-once memories. To appear in *Proceedings of the 34th International Colloquium on Automata, Languages and Programming*, 2007.

[17] M. Naor and V. Teague. Anti-persistence: History independent data structures. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, pages 492–501, 2001.

[18] N. Nisan and A. Ta-Shma. Extracting randomness: A survey and new constructions. *Journal of Computer and System Sciences*, 58(1):148–173, 1999.

[19] R. L. Rivest and A. Shamir. How to reuse a "write-once" memory. *Information and Control*, 55(1-3):1–19, 1982. A preliminary version appeared in *Proceedings of the 14th Annual ACM Symposium on Theory of Computing*, pages 105–113, 1982.

[20] M. Sipser. Expanders, randomness, or time versus space. *Journal of Computer and System Sciences*, 36(3):379–383, 1988.

[21] A. Ta-Shma. Storing information with extractors. *Information Processing Letters*, 83(5):267–274, 2002.

[22] A. Ta-Shma, C. Umans, and D. Zuckerman. Loss-less condensers, unbalanced expanders, and extractors. In *Proceedings of the 33rd Annual ACM Symposium on Theory of Computing*, pages 143–152, 2001.