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 study 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. 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.

We also demonstrate a connection between secure vote storage mechanisms and 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 in *Proceedings of the 34th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 303–315, 2007.

*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

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. Our motivation emerges from the task of designing vote storage mechanisms, recently studied by Molnar, Kohno, Sastry and Wagner [20]. The setting considered by Molnar et al. is that of an electronic voting machine in a polling station. In a typical election, the machine is set up by local election officials. Voters are then allowed to cast their ballots. Finally, the 'polls are closed' by the election officials (after which no additional ballots may be cast), and the results transmitted to a voting center. The machines themselves may also be used to audit or verify the results.

This setting is an acute example of a hostile environment for voting machines: an adversary attempting to corrupt the election results may also be a legitimate voter, an election official, or even one of the voting machine developers. 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. The attacker may attempt to change, add or delete votes, or merely to learn how others voted (in order to buy votes or coerce voters). Without a 'secure' vote storage mechanism, such an adversary may be able to undetectably tamper with the voting records or compromise voter privacy.

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. In the vote storage context, think of 'elements' as ballots, K as the number of voters and of N as the number of possible ballot states (e.g., if there are 10 two-candidate races, there are $N = 2^{10}$ possible ballot states; alternatively, if there is one race where write-in candidates are allowed, N would be the number of possible candidate names). Our mechanism supports insert operations, membership queries, and enumeration of all stored elements[1]. While 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. We design a storage mechanism which is deterministic, history-independent, and tamper-evident.

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 covert channel: As multiple valid representations for the same abstract state exist, a maliciously designed 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

[1] We note that for vote storage mechanisms it is sufficient to support only insert operations and enumeration of all stored elements

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 (e.g., see [1, 2]). Informally, we consider a period of activity after which the memory representation of the data is revealed to the adversary. The data structure is history-independent 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 3.1).

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. [20], 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. Consider, for example, 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. This encoding has the property that 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 (which is physically impossible when using a write-once memory). 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).

1.1 Our Contributions

We construct an efficient, 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. Previous constructions were either much less efficient (required $\Theta(K^2)$ memory), randomized, or employed cryptographic techniques that require secure key storage (making them vulnerable to various side-channel and hardware attacks). Unless stated otherwise, throughout the paper we refer to the amount of allocated memory as the number of allocated memory *words*, each of length $\log N$ bits, and assume that writing and reading a memory word can be done in constant time. Our main result is the following:

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 $O(K \cdot \mathrm{polylog}(N))$.*
2. *The amortized insertion time and the worst-case look-up time are $O(\mathrm{polylog}(N))$.*

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 $O(\log^2 N \cdot \log K)$.*
3. *The worst-case look-up time is $O(\log N \cdot \log K)$.*

In order to evaluate the security of our mechanism we focus on the main security goals of vote storage mechanisms [20], and formalize a threat model. Such a model should specify both the computational capabilities of the adversary (in this paper we consider computationally unbounded adversaries), and the type of access that the adversary has to the mechanism. Our threat model is described in Section 3.2. Informally, we consider two types of adversaries: *post-election adversaries* that gain access to the mechanism at the end of the election process, and *lunch-time adversaries* that gain access to the mechanism at several points in time during the election process. For each type of adversaries we consider two levels of access to the mechanism: read-only access, and read-write access.

We show that our mechanism provides the highest level of security against post-election adversaries with read-write access, and against lunch-time adversaries with read-only access. Unfortunately, our mechanism turns out to be insecure against lunch-time adversaries with read-write access. Specifically, it does not guarantee tamper-evidence against such adversaries. We prove, however, that such a vulnerability is not specific for our construction, but is inherent in any mechanism that uses significantly less than K^2 bits of storage. In fact, we provide a complete characterization of the class of deterministic, history-independent and write-once mechanisms that do enjoy such a level of security. Informally, we show that any such mechanism stores the elements according to a superimposed code [17]. The following theorem then follows from known lower bounds and upper bounds on superimposed codes [11, 12, 25]:

Theorem 1.3. *Any deterministic, history-independent, and write-once mechanism for storing a set of at most K elements from a universe of size N which is tamper-evident against a lunch-time adversary with read-write access uses $\Omega\left(\frac{K^2}{\log K} \cdot \log N\right)$ bits of storage. Moreover, there exists such an explicit mechanism that uses $O(K^2 \log^2 N)$ bits of storage.*

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 individually 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 (with the exception that a station which successfully transmits halts, and waits for the algorithm to terminate). The efficiency measure for conflict resolution algorithms is the total number of steps it takes to resolve conflicts in the worst case (where the worst case refers to the maximum over all possible sets of conflicting stations).

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 [18] 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 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 [18] up to poly-logarithmic factors. More specifically, we prove the following theorem:

Theorem 1.4. *For every N and K there exists an explicit, deterministic, and non-adaptive algorithm that resolves any K conflicts among N stations in $O(K \cdot \mathrm{polylog}(N))$ steps.*

Paper organization. The rest of the paper is organized as follows. In Section 2 we review related work. Section 3 contains some essential definitions and a formal description of our main security goals and threat model. In Section 4 we present our construction of the storage mechanism, which we then analyze in Section 5. The analysis includes, in addition to an evaluation of the soundness, performance, and security guarantees of our construction, a characterization of the class of mechanisms that are deterministic, history-independent, and write-once and provide tamper-evidence against a lunch-time adversary with read-write access. In Section 6 we provide constructions of the bipartite graphs that serve as the main building block of our storage mechanism. Finally, in Section 7 we show that our technique can be adapted to devise a deterministic and non-adaptive conflict resolution algorithm.

2 Related Work

The problem of constructing history-independent data structures was first formally considered by Micciancio [19], 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 [22] and consider a stronger notion — data structures whose *memory representation* does not leak information (see Section 3.1 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. Some of their results were recently improved by Blelloch and Golovin [6] and by Naor et al. [21] who managed to support delete operations as well while still guaranteeing the strongest form of history independence.

In the context of write-once memories, Rivest and Shamir [24] 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 [16] 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. [20] 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 [22], 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. [20]. 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. We note that when dealing with a small universe of elements (for example, an election with only two candidates), a better solution is to pre-allocate memory to store a bounded unary counter for each element. However, this may not be suitable for elections in cases where write-in candidates are allowed (as is common in the U.S.) or when votes are subsets or rankings (as is common in many countries).

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 covert channels: a maliciously designed 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. The constructions of Molnar et al. achieved tamper-evidence by exploiting the properties of write-once memories. Bethencourt, Boneh and Waters [5] took a different approach to designing a history-independent tamper-evident storage mechanism. They developed a signature scheme for signing sets of elements with two important properties: the order in which elements were added to the set cannot be determined from the signature, 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).

3 Definitions and Threat Model

3.1 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 3.1. 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 [22], 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. [15] and by Buchbinder and Petrank [7]. Since we deal only with deterministic data structures, in our setting the definitions are equivalent.

3.2 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 security goals and threat model in terms of a vote storage mechanism. In an actual voting scenario, casting a ballot corresponds to an `Insert` operation. In the simplest form of voting systems, the element inserted is the chosen candidate's name. In more complex voting systems, however, the inserted element may be a ranking or a subset of the candidates, an encrypted 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.

The main security goals we would like our storage mechanism to achieve are the following[2]:

1. **Tamper-evidence**: Any modification of votes after they were cast should 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.

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). The extent to which each of the above security goals can be satisfied by our mechanism depends on the assumed adversarial access. We consider two types of adversaries: *post-election adversaries* that gain access to the mechanism at the end of the election process, and *lunch-time adversaries* that gain access to the mechanism at several points in time during the election process. For each type of adversaries we consider two levels of access to the mechanism: read-only access, and read-write access. In Section 5.2 we evaluate the security of our mechanism according to the above security goals and threat model.

[2]For simplicity we focus on the main and most relevant security goals. We refer the reader to the work of Molnar et al. [20] for a more detailed list.

4 The Construction

4.1 Overview

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. [20], and are based on an observation by Naor and Teague [22], 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*[3]. Our global strategy will map all the elements 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

[3]The definition is motivated by the notion of bipartite unique-neighbor expanders presented by Alon and Capalbo [3].

the same left set L. Each graph will enable us to store a fraction of the elements in S. Formally, we define:

Definition 4.1. 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 4.2. 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 6.

4.2 Details

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 4.1, 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 the mechanism of Molnar et al. [20] by using the encoding discussed in the introduction (specifically, the seal operation concatenates to the current content of the memory the binary representation of the Hamming weight of its complement). The operations $\texttt{Insert}(x, T_i)$, $\texttt{LookUp}(x, T_i)$, and $\texttt{RetrieveAll}()$ are described in Figure 1.

5 Analysis of the Construction

5.1 Soundness and Performance

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

`Insert`(x, T_i):

for all neighbors y of x in the graph G_i **do**
  if $T_i[y] = *$ **then**
    Continue to the next neighbor of x
  else if $|T_i[y]| < \ell_i$ **then**
    Store x in the copy-over list $T_i[y]$
  else
    for all x' in $T_i[y]$ such that x' does not appear in any other list in T_i **do**
      Execute `Insert`(x', T_{i+1})
    Set $T_i[y] \leftarrow *$ // erase the memory blocks of $T_i[y]$
if x was not stored in any copy-over list in the previous step **then**
  Execute `Insert`(x, T_{i+1})

`LookUp`(x, T_i):

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

`RetrieveAll()`:

for all tables T_i **do**
  for all copy-over lists $T_i[y]$ **do**
    if $T_i[y] \neq *$ **then**
      Output all elements of $T_i[y]$ that have not yet been output

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

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.

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 is 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 (note that although the elements of S'_i are *inserted* into table T_i this does not necessarily mean that they will eventually be stored 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 5.2. *For every set $S \subseteq [N]$ of size at most K, for every insertion order of its elements, and for every $0 \le i \le t$, the number of* `Insert`$(\cdot, T_i)$ *calls is at most K_i. In particular, if there exists an $\alpha > 0$ such that $\alpha_i \ge \alpha$ for every G_i, then setting $t = \lceil \frac{\ln K}{\alpha} \rceil$ 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 4.2 implies there are at least $\alpha_i K_i$ 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 $x \in S_i$ 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. 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 $\alpha_i \ge \alpha$ for every G_i, then in particular the number of `Insert`$(\cdot, T_t)$ calls is at most

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

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 5.3. *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 $\frac{1}{K} \cdot \left(\sum_{i=0}^{t} |R_i| \cdot \ell_i^2\right) + \sum_{i=0}^{t} D_i^2 \cdot \ell_i^3$.*
3. *The* worst-case *look-up time is at most $2 \cdot \sum_{i=0}^{t} D_i \cdot (\log \ell_i + 1)$.*

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 $\sum_{i=0}^{t} |R_i| \cdot \ell_i^2$.

In order to bound the amortized insertion time, we consider the number of write operations and the number of read operations separately. Since the storage strategy is write-once, then the total number of write operations when storing K elements is upper bounded by the amount of memory in which the elements are stored. Therefore, the amortized number of write operations per insertion is at most $\frac{1}{K} \cdot \left(\sum_{i=0}^{t} |R_i| \cdot \ell_i^2\right)$. We bound the amortized number of read operation as follows: Each element is inserted into at most t tables, and into D_i entries of each table. In the worst case, when an element is inserted into an overflowing copy-over list, we scan the current table for all the ℓ_i elements that are stored in the overflowing list, which can be done in $D_i \cdot \ell_i^2$ read operations for every such element. Therefore, the amortized number of read operations is at most $\sum_{i=0}^{t} D_i^2 \cdot \ell_i^3$.

Finally, 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 (and therefore can be searched in time $\lceil 2\log \ell_i \rceil$). Therefore, the worst-case look-up time is at most $2 \cdot \sum_{i=0}^{t} D_i \cdot (\log \ell_i + 1)$. ■

Theorems 1.1 and 1.2 now follow by instantiating the mechanism with the bounded-neighbor expanders from Corollary 6.7 and Theorem 6.1, respectively.

Proof of Theorem 1.1. When the sequence of graphs $G_0, \dots, G_t$ are constructed according to Corollary 6.7 with $\epsilon = 1/2$, we have that every G_i is a (K_i, α_i, ℓ_i)-bounded-neighbor expander, such that

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

for some constant $c > 0$ and $D = \text{polylog}(N)$. Therefore, by Lemma 5.2, we can set $t = \left\lceil \frac{\ln K}{\alpha} \right\rceil$, where $\alpha = \frac{c}{4D\log^3(N)}$. Now, Lemma 5.3 states that the total amount of allocated memory is

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

Thus, the required memory allocation is $O(K \cdot \text{polylog}(N))$. Very similarly to the above calculation, Lemma 5.3 further implies that the amortized insertion time and the worst-case look-up time are $O(\text{polylog}(N))$. ■

Proof of Theorem 1.2. When the sequence of graphs $G_0, \dots, G_t$ are constructed according to Corollary 6.1, we have that every G_i is a $(K_i, 1/2, 1)$-bounded-neighbor expander, where $K_i = K/2^i$, $|R_i| = c_1 \cdot K_i \log(N/K_i)$ and $D_i = c_2 \cdot \log(N/K_i)$ for some constants $c_1, c_2 > 0$. Now, Lemma 5.3 states that:

1. The total amount of allocated memory is

$$
\begin{aligned}
\sum_{i=0}^{t} |R_i| \cdot \ell_i^2 &= c_1 \cdot \sum_{i=0}^{t} \frac{K}{2^i} \cdot \log\left(\frac{N \cdot 2^i}{K}\right) \\
&= c_1 \cdot \sum_{i=0}^{t} \frac{K}{2^i} \cdot \log\left(\frac{N}{K}\right) + c_1 \cdot \sum_{i=0}^{t} \frac{i \cdot K}{2^i} \\
&\leq 2c_1 K \log\left(\frac{N}{K}\right) + 2c_1 K \\
&= O\left(K \log\left(\frac{N}{K}\right)\right) \quad .
\end{aligned}
$$

2. The amortized insertion time is at most

$$
\begin{aligned}
\frac{1}{K} \cdot \left(\sum_{i=0}^{t} |R_i| \cdot \ell_i^2\right) + \sum_{i=0}^{t} D_i^2 \cdot \ell_i^3 &= \frac{1}{K} \cdot \left(2c_1 K \log\left(\frac{N}{K}\right) + 2c_1 K\right) + c_2 \cdot \sum_{i=0}^{t} \log^2\left(\frac{N \cdot 2^i}{K}\right) \\
&= O\left(\log^2 N \cdot \log K\right) \quad .
\end{aligned}
$$

3. The worst-case lookup time is at most

$$
\begin{aligned}
2\sum_{i=0}^{t} D_i &= 2c_2 \cdot \sum_{i=0}^{t} \log\left(\frac{N \cdot 2^i}{K}\right) \\
&= O(\log N \cdot \log K) \quad .
\end{aligned}
$$

■

5.2 Security Evaluation and Characterization

In this section we evaluate the security of our mechanism according to the security goals and threat model which we formalized in Section 3.2 in terms of vote storage mechanisms. In addition, we characterize the class of mechanisms that are deterministic, history-independent, and write-once and provide tamper-evidence against a lunch-time adversary with read-write access. Recall that our main security goals are to guarantee tamper-evidence, privacy, and robustness, and we consider two types of adversaries: post-election adversaries and lunch-time adversaries.

5.2.1 Security Against Post-Elections Adversaries

We first consider a post-election adversary that has read-only access to the mechanism. In this case, tamper-evidence and robustness are trivially satisfied since the adversary does not modify the records. Privacy is guaranteed due to the history-independence of the mechanism (see Lemma 5.1).

Now consider a post-election adversary that has read-write access to the mechanism. In this case, tamper-evidence is guaranteed due to the write-once memory: at the end of the election process, the records are sealed using the encoding suggested by Molnar et al. [20], and therefore it is impossible to undetectably modify the records. Privacy is again guaranteed by the history-independence property of the mechanism. Robustness, however, cannot be satisfied in such a case since the adversary can simply erase the records by flipping all bits to 1.

5.2.2 Security Against Lunch-Time Adversaries

Consider a lunch-time adversary that has read-only access to the mechanism. That is, the adversary obtains the memory representation of the mechanism at several points in time during the election process. As in the case of a read-only post-election adversary, tamper-evidence and robustness are trivially satisfied. Privacy is guaranteed by the strong history-independence property of the mechanism. More specifically, each time the adversary obtains the memory representation of the mechanism, the only information that is leaked is the set of elements inserted since the previous time the memory representation was revealed. This is the highest possible level of privacy against such an adversary.

We now turn to consider a lunch-time adversary that has read-write access to the mechanism. That is, the adversary gains read-write access to the mechanism at several points in time during the election process. In such a case, our mechanism still provides the highest possible level of privacy, exactly as in the case of a read-only lunch-time adversary. Robustness, however, is impossible to guarantee in such a case since the adversary can erase the records.

The task of guaranteeing tamper-evidence against a read-write lunch-time adversary turns out to be more complicated. When considering such an adversary, the best we can hope for is a guarantee about operations that took place before the attack (i.e., before the adversary gained control). The adversary should not be able to undetectably delete votes that were previously cast. Unfortunately, our mechanism does not guarantee this property, and in fact we manage to provide a complete characterization of the class of deterministic, history-independent and write-once mechanisms that do guarantee this property. We show that any such mechanism guarantees this property if and only if it stores the elements according to an $(N, K+1)$-superimposed code [17]. A known lower bound on superimposed codes (see, for example, [12, 25]) implies that $\Omega\left(\frac{K^2}{\log K} \cdot \log N\right)$ memory bits are required in order to store K elements, whereas our mechanism uses only $O(K \cdot \mathrm{polylog}(N))$ bits. Moreover, using known explicit constructions of superimposed codes we show that $O(K^2 \log^2 N)$ bits suffice, and this proves Theorem 1.3.

The reason that our mechanism is not tamper-evident against read-write lunch-time adversaries is as follows. There exist two legal memory configurations, C_1 and C_2, with the following three properties: (1) C_1 is obtained by inserting some element x, (2) C_2 is obtained by inserting a set of K elements $x_1, \ldots, x_K$ that are all different than x, and (3) C_1 is "contained" in C_2 in the sense that for every index i, if the i-th bit of C_1 is set to 1 then also the i-th bit of C_2 is set to 1. The existence of such memory configurations C_1 and C_2 enables an adversary to mount the following attack: the adversary gains control over the mechanism with memory representation C_1 (i.e., only the element x was inserted), and simply changes it to C_2 (note that this is possible due to property (3)). Now, the memory representation corresponds to the set of elements $x_1, \ldots, x_K$, and there is no trace for the fact that x was ever inserted. That is, the adversary managed to delete an element that was inserted before the adversary gained control, and this is clearly undetectable since the new memory representation is legal. A superimposed code has exactly the property that prevents such a situation: a codeword cannot be covered by any small number of other codewords.

A construction using superimposed codes. We present a simple construction of a deterministic, history-independent and write-once mechanism, which requires $O(K^2 \log^2 N)$ memory bits in order to store an increasingly growing set of at most K elements taken from the universe $[N]$. The mechanism maps the elements of the universe into entries of a table according to a superimposed code. More specifically, given N and K, a binary superimposed code of size N guarantees that any codeword is not contained in the bit-wise or of any other $K-1$ codewords. In what follows for binary strings $y = y_1 \cdots y_n \in \{0,1\}^n$ and $y' = y'_1 \cdots y'_n \in \{0,1\}^n$ we use $y \subseteq y'$ for denoting that for every $1 \leq i \leq n$ it holds that $y_i \leq y'_i$, and we use $y \nsubseteq y'$ for denoting that there exists an index $1 \leq i \leq n$ such that $y_i > y'_i$ (i.e., we naturally interpret y and y' as subsets of $\{1,\ldots,n\}$). We use the following result of Erdös, Frankel and Füredi [11].

Theorem 5.4 ([11]). *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 $C(x_1) \nsubseteq \bigvee_{i=2}^{\ell} C(x_i)$.*

Given such a code $C : [N] \to \{0,1\}^d$ with $\ell = K+1$, the mechanism consists of a table T containing d entries, denoted $T[1],\ldots,T[d]$. In order to insert an element x, we store x in all entries $T[i]$ for which $C(x)_i = 1$. If an entry is already occupied, it is permanently deleted. The superimposed code guarantees that if at most K elements are inserted, then each element will be successfully stored (that is, for each element there exists an entry which is unique for the element). The mechanism is clearly history-independent and write-once. Moreover, the superimposed code guarantees tamper-evidence against a read-write lunch-time adversary: an existing element cannot be deleted unless more than K elements are inserted.

A lower bound. We prove a lower bound on the amount of memory bits used by any mechanism which is deterministic, history-independent, write-once, and guarantees tamper-evidence against a read-write lunch-time adversary. We show that any such mechanism which uses d bits of memory can be used to define an $(N, K+1)$-superimposed code $C : [N] \to \{0,1\}^d$. Thus, the above mentioned lower bound for superimposed codes implies that $d = \Omega\left(\frac{K^2}{\log K} \cdot \log N\right)$.

Given such a mechanism, we define a mapping $C : [N] \to \{0,1\}^d$ as follows. For any $x \in [N]$ let $C(x)$ denote the memory representation of the mechanism when it contains the singleton $\{x\}$. In what follows we argue that C is an $(N, K+1)$-superimposed code. First, we extend the mapping C for *sets* of elements: for any set $S \subset [N]$ denote by $C(S)$ the memory representation of the mechanism when it contains the set S. We note that the mapping C is well-defined since a mechanism which is deterministic and history-independent must have a unique representation for each set of elements. In addition, the write-once property implies that C is monotone. That is, for any two sets $S_1, S_2 \subseteq [N]$ such that $S_1 \subseteq S_2$ it holds that $C(S_1) \subseteq C(S_2)$ (that is, $C(S_2)$ can be obtained from $C(S_1)$by only flipping bits from 0 to 1).

Assume for the purpose of deriving a contradiction that C is not an $(N, K+1)$-superimposed code. Then there exist distinct $x_1,\ldots,x_{K+1} \in [N]$ for which $C(x_1) \subseteq \bigvee_{i=2}^{K+1} C(x_i)$. Notice that this implies that $C(x_1) \subseteq C(\{x_2,\ldots,x_{K+1}\})$. Consider an attack in where the adversary gains control over the mechanism when it contains the singleton $\{x_1\}$. At this point the adversary can modify the memory representation to $C(\{x_2,\ldots,x_{K+1}\})$ by flipping bits from 0 to 1, and obtain the unique memory representation of the set $\{x_2,\ldots,x_{K+1}\}$. That is, the adversary managed to undetectably delete x_1. This yields a contradiction to the assumed security of the mechanism, and therefore the mapping C is an $(N, K+1)$-superimposed code.

6 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 [26] is in fact a bounded-neighbor expander.

6.1 A Non-Constructive Proof

We prove the following theorem:

Theorem 6.1. *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.

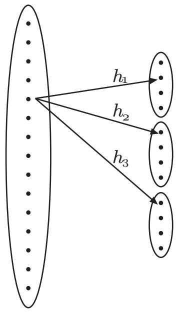

Figure 2: The constructed bounded-neighbor expander for the case $t = 3$.

Lemma 6.2. *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 6.3. *For every N and $K \leq N$, 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. Fix N and $K \leq N$. We apply the probabilistic method and show that with positive probability over the random choice of such a family $\mathcal{H}$ it holds 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 $[3K]$. More specifically, consider the experiment of constructing the family $\mathcal{H}$ by choosing uniformly and independently at random a collection of

$$|\mathcal{H}| = \left\lceil \frac{48}{K} \cdot \log \binom{N}{K} \right\rceil + 1 = O\left(\log\left(\frac{N}{K}\right)\right)$$

functions $h : [N] \to [3K]$. Then Lemma 6.2 implies that for every fixed set $S \subseteq [N]$ of size K, the probability that there is no function $h \in \mathcal{H}$ whose restriction to S maps at least $K/2$ elements of S to unique elements of $[3K]$ is at most $\exp(-K/48 \cdot |\mathcal{H}|)$. Therefore, the probability over the choice of $\mathcal{H}$ that there exists a set $S \subseteq [N]$ of size K for which there is no function $h \in \mathcal{H}$ whose restriction to S maps at least $K/2$ elements of S to unique elements of $[3K]$ is at most

$$\binom{N}{K} \exp\left(\frac{K}{48} \cdot |\mathcal{H}|\right) < 1 \ .$$

■

6.2 An Explicit Construction

We provide an explicit construction of bounded-neighbor expanders by showing that any disperser is a bounded-neighbor expander. Dispersers [26] 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 [23] for a comprehensive survey). We now formally define dispersers, and then show that any disperser is a bounded-neighbor expander.

Definition 6.4. 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 6.5. *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 $\ell = \lceil 1/\alpha \rceil$.*

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). Therefore, we can focus on 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

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

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 6.5 can be instantiated with the following disperser construction of Ta-Shma, Umans and Zuckerman [28].

Theorem 6.6 ([28])**.** *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 $D = \mathrm{polylog}(N)$.*

Corollary 6.7. *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 $D = \mathrm{polylog}(N)$, and $\alpha = \frac{(1-\epsilon)|R|}{2DK}$.*

An alternative approach for constructing bounded-neighbor expanders is by using lossless condensers[4]. This approach guarantees constant α and very small ℓ, but larger $|R|$. The recent construction of Guruswami, Umans and Vadhan [14] 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 $K = \mathrm{polylog}(N)$.

7 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 individually 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.

[4]We note that unbalanced expanders have been already considered for storing sets of elements by Buhrman, Miltersen, Radhakrishnan and Venkatesh [8] and by Ta-Shma [27] with the property that membership queries can be answered by querying just one bit.

Several deterministic adaptive solutions are known. Capetanakis's tree algorithms [9, 10], that resolve conflicts in $O(K \log(N/K))$ steps, were devised almost three decades ago. Greenberg and Winograd [13] showed that any deterministic algorithm must run for $\Omega(K(\log N)/\log K)$ steps. In 1985, Komlós and Greenberg [18] 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.

7.1 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 [17]. 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. For our algorithm we use the superimposed code of Erdös, Frankel and Füredi [11] whose properties we stated in Theorem 5.4, and note that any other superimposed code with similar asymptotic guarantees can be used.

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.

7.2 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 $C_i(x_1) \nsubseteq \bigvee_{j=2}^{\ell_i} C_i(x_j)$.

The algorithm runs in a sequence of intervals $I_0, \ldots, I_t$. Each interval I_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. Theorem 1.4 is proved by instantiating the algorithm with the explicit family of bounded-neighbor expanders constructed in Section 6. The proof is almost identical to the proof of Theorem 1.1, and is omitted.

Lemma 7.1. *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.*

Proof. We prove the first part of the lemma, and the second part of the lemma follows directly from the description of the algorithm, the first part of the lemma, and Theorem 5.4.

We prove the first part of the lemma by induction on i. Denote by $S \subseteq [N]$ the set of K conflicting stations, and for every $1 \leq i \leq t$ denote by $S_i \subseteq [N]$ the set of stations that are still active at the beginning of interval I_i. Then $S_0 = S$, which implies that $|S_0| \leq K_0 = K$.

Suppose now that $|S_i| \leq K_i$, i.e., that the number of active stations at the beginning of interval I_i is at most K_i. 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 successfully transmit during the sub-interval $I_i[y]$ due to the property of the superimposed code C_i. This implies that the number of stations which will remain active at the beginning of interval I_{i+1} (i.e., the size of the set S_{i+1}) 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 4.2 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 $x \in S_i$ that has an ℓ_i-degree neighbor with respect to S'_i, also has an ℓ_i-degree neighbor with respect to S_i (this is the same neighbor). 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. Therefore, $|S_{i+1}| \leq K_{i+1}$ and the lemma follows. ■

8 Concluding Remarks

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 retrieve 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.

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 [22], 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 6 does not achieve the parameters that one can hope for according to Theorem 6.1. 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.

Optimal monotone encoding. 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. This can be alternatively formulated as the problem of finding an optimal monotone encoding: 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 such encoding can be translated into a write-once strategy that requires a memory of size M bits. This problem was posed in a preliminary version of our work, and was recently solved by Alon and Hod [4], who provided a non-constructive proof showing that $M = O(K \log(N/K))$.

Acknowledgments

The authors would like to thank Ronen Gradwohl, David Wagner, and the anonymous referees for many useful comments.

References

[1] Readers 'declassify' US document. BBC News, May 2005. Available at `http://news.bbc.co.uk/1/hi/world/europe/4506517.stm`.

[2] AT&T leaks sensitive info in NSA suit. CNET News, May 2006. `http://news.cnet.com/AT38T-leaks-sensitive-info-in-NSA-suit/2100-1028_3-6077353.html`.

[3] 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.

[4] N. Alon and R. Hod. Optimal monotone encodings. In *Proceedings of the 35th International Colloquium on Automata,Languages and Programming*, pages 258–270, 2008.

[5] 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.

[6] G. E. Blelloch and D. Golovin. Strongly history-independent hashing with applications. In *Proceedings of the 48th Annual IEEE Symposium on Foundations of Computer Science*, pages 272–282, 2007.

[7] N. Buchbinder and E. Petrank. Lower and upper bounds on obtaining history-independence. *Information and Computation*, 204(2):291–337, 2006.

[8] H. Buhrman, P. B. Miltersen, J. Radhakrishnan, and S. Venkatesh. Are bitvectors optimal? *SIAM Journal on Computing*, 31(6):1723–1744, 2002.

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

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

[11] 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.

[12] Z. Füredi. On r-cover-free families. *Journal of Combinatorial Theory, Series A*, 73(1):172–173, 1996.

[13] 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.

[14] 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*, pages 96–108, 2007.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

[20] 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. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 365–370, 2006.

[21] M. Naor, G. Segev, and U. Wieder. History-independent cuckoo hashing. In *Proceedings of the 35th International Colloquium on Automata, Languages and Programming*, pages 631–642, 2008.

[22] 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.

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

[24] R. L. Rivest and A. Shamir. How to reuse a "write-once" memory. *Information and Control*, 55(1-3):1–19, 1982.

[25] M. Ruszinkó. On the upper bound of the size of the r-cover-free families. *Journal of Combinatorial Theory, Series A*, 66(2):302–310, 1994.

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

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

[28] A. Ta-Shma, C. Umans, and D. Zuckerman. Lossless condensers, unbalanced expanders, and extractors. *Combinatorica*, 27(2):213–240, 2007.