Compression And Decompression Under FHE Using Error-Correcting Codes and Copy-And-Recurse

Adi Akavia*
adi.akavia@gmail.com
University of Haifa
Israel
TU Darmstadt
Germany

Hayim Shaul
hayim.shaul@gmail.com
IBM Research
Israel

Ofer Shayevitz
ofersha@gmail.com
Tel Aviv University
Israel

Abstract

Compression has been a fundamental problem in computer science for decades. Simply put, we want to represent a low-entropy vector v of size n with less than n elements so that v can be reconstructed (decompressed) from the shorter representation. Since compressed vectors require less storage and less communication, compression algorithms are part of almost every digital system.

When the vector is encrypted with fully homomorphic encryption (FHE) the problem becomes significantly harder. Some research (e.g., [TCHES'19, CCS'21, EuroCrypt'23 ,USENIX'24]) have considered the problem of compressing an encrypted vector but they all assumed the decompression step happens in cleartext. This is a significant restriction. For example, any system with an untrusted agent that needs to receive data and analyze it cannot use existing compression algorithms.

In this paper, we give the first (to the best of our knowledge) non-trivial compression-decompression algorithms that are both FHE-friendly. Our algorithms use the copy-and-recurse technique together with the known duality between compression and error-correcting codes. Our experiments show that our decompression algorithm is faster than the folklore decompression algorithm. This is useful in systems with an agent-in-the-middle that is bounded by communication and by computation.

Keywords

Fully Homomorphic Encryption, Secure Computation, Compression, Error-Correcting Codes

1 Introduction

Compression is a problem that has been researched since the early days of computer and digital systems. Simply put, we want to represent a low-entropy vector $\mathbf{v}$ of size n with fewer than n elements in a way that we can still reconstruct $\mathbf{v}$. The motivation is to reduce storage or communication, which is useful in a wide range of applications.

When the elements of $\mathbf{v}$ are encrypted with fully homomorphic encryption (FHE), $\mathbf{v}$ can still be compressed (without being decrypted) and then decompressed (still without being decrypted). The problem of compression under FHE was discussed in [4] and later in other works, e.g., [8, 13, 16]. The motivation they considered was a client-server system where the server receives an encrypted query from the client, computes (under FHE) a reply that is a vector which is known to have low entropy, compresses it (still under FHE) and then sends it to the client. In all previous works the decompression was too inefficient to run under FHE and the client first **decrypts** the compressed vector before decompressesing it in cleartext. This solution does not fit systems where an untrusted agent acts on behalf of the client. In these system the agent communicates with one or more servers and analyzes their replies before returning it to the user. See Figure 1 for an illustration. Without efficient decompression under FHE, client-agent-server systems, need to either: (1) refrain from compressing data between the server and the agent; (2) trust the agent with the secret key; or (3) add rounds of communications between the client and the agent to provide the latter with a decompression service. All options are inefficient or insecure (the client or the agent learns intermediate values).

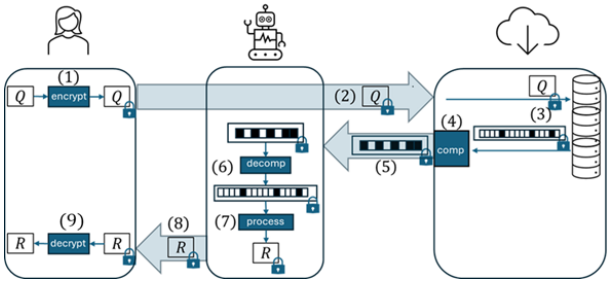

Figure 1: A client-agent-server system using compression and decompression under FHE. A client (left) encrypts (1) a query and sends it (2) to the server (right) over an expensive link. The server queries the database and gets an encrypted result which is a sparse indicator vector. The result is compressed under FHE (4) and sent back to an agent (middle) (5). The agent decompresses the results under FHE (6) and then processes it (7). The processed result is sent to the client (8) who then decrypts it (9).

The case of s-sparse vectors. In this paper we consider s-sparse vectors (i.e., having at most s nonzero elements). This type of vectors is especially interesting because many low-entropy vectors can be reduced to be s-sparse. Specifically, if

*The first author was supported in part by the National Research Center for Applied Cybersecurity ATHENE.

there is a model, M, that predicts the ith element from the preceding elements, $v_i = M(v_1, \ldots, v_{i-1})$, with probability $> 1 - s/n$, then the difference-vector between the predictions of M and $\mathbf{v}$ is s-sparse. Another motivation for s-sparse vectors comes from hash tables. There, we insert s elements into a hash table, H, of size n by mapping each element to a single slot in the table. *Collision* is when 2 elements are mapped to the same slot which leads to a failure. Increasing n decreases the probability for collision (failure) but come with the cost of an increased table size which is significant when the H needs to be stored or communicated. Using compression we can increase the table size while keeping these costs low. We note that in some variations of hash tables (e.g., cuckoo hash), an element is mapped to multiple slots (but inserted to only one). While these variations decrease the probability for failure without increasing the table size, they come with a cost that searching in them, or intersecting 2 tables (i.e., finding elements that appear in both tables) is more time consuming.

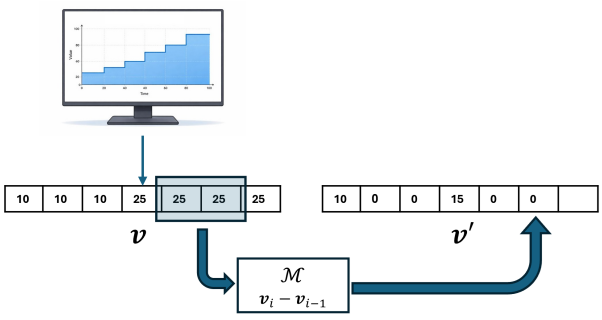

Figure 2: Values from a monitor system change rarely. A simple model predicts $\mathbf{v}_i$ to be the same as $\mathbf{v}_{i-1}$. The difference between the elements of $\mathbf{v}$ and the model prediction is a sparse vector.

We give a compression algorithm and a matching decompression algorithm for s-sparse vectors that are both FHE-friendly. This allows for agentic systems as described above. To the best of our knowledge, we are the first to provide an efficient solution for compression and decompression under FHE.

Our improved result comes from using a new technique called copy-and-recurse that was recently introduced in [22, 23] and from the well-known observation that compression and error-correcting codes (ECC) are related.

Copy-and-recurse is used to convert a cleartext algorithm $\mathcal{A}_{clear}$ that traverses a tree to an algorithm $\mathcal{A}_{FHE}$ that efficiently traverses that tree under FHE. The runtime complexity is $Time(\mathcal{A}_{FHE}) = Time(\mathcal{A}_{clear}) + O(n)$, where n is the number of leaves. Specifically, the cumulative work done at nodes that $\mathcal{A}_{clear}$ visits stays the same (asymptotically) under FHE. In our work, we keep the elements of the input vector in the leaves of a binary tree and use copy-and-recurse to traverse the s paths that lead to the nonzero leaves. The output of the compression algorithm is a compact description of these paths. The decompression algorithm uses this description to reconstruct the tree.

The duality between **error-correcting codes** and compression is a well-known observation [20]. Nevertheless, all previous works under FHE used tailored solutions. The first work [4] used ideas from group-testing to compress a s-sparse binary vector. They also mention that compressing a s-sparse vector over $\mathbb{Z}_p$ translates to compressing $\log p$ s-sparse binary vectors.

In [13] the authors showed 2 tailored solutions for compressing a binary vector. One of their solutions was described with arithmetic bloom filter which in the terminology of ECC is a linear code. Their second solution had a more explicit description. In the terminology of ECC they used a Vandermonde matrix for the generating matrix in the linear code. In [16] the authors gave 2 more tailored solutions. The first solution defines a polynomial $P(x) = \prod a_i x^i$, where $(a_1, \ldots)$ is the sparse input. Then they used FFT to efficiently encode $P(x)$. In the terminology of ECC this is equivalent to Reed-Solomon code. Their second solution uses invertible Bloom lookup tables (IBLT) [19].

Another line of work considered a related problem called *batch-PIR*. Here, the goal is to encode only the nonzero values (and not their indexes). It is easy to see that given a binary vector $\mathbf{a} = (a_1, a_2, \ldots, a_n)$ then using batch-PIR on $\mathbf{a}' = (1 \cdot a_1, \ldots, n \cdot a_n)$ gives the indexes of the nonzero elements thus encoding $\mathbf{a}$.

Batch-PIR was recently studied by [8] who improved the work of [14] which used *sparse random linear coding* [26]. In the terminology of ECC, they applied a linear code by multiplying $\mathbf{a}$ by a generating matrix. Specifically, they used *random band matrices* that have better performance than random matrices.

We think that explicitly stating the duality between compression and ECC (in the context of FHE) is interesting on its own because it opens the door to the rich literature that already exists for ECC. In this sense, we hope future work will (1) revisit and use the literature from ECC to improve previous work; (2) derive new compression algorithms for FHE, inspired by ECC codes (e.g., Turbo codes [7], Raptor codes [25]).

Low-rate codes - a new regime. In addition to the above, the compression-ECC duality opens a new regime in information theory. More specifically, the dual of compressing a s-sparse vector of size n, with $s \ll n$, is a low-rate code. In cleartext this is trivial and has received little attention. The restrictions of FHE, which makes this problem non-trivial, motivates research in this regime.

1.1 Our Contribution

We summarize our contribution:

- **Compression and decompression algorithms under FHE.** We propose a novel compression algorithm and a matching decompression algorithm that are both FHE friendly.

- **Compression - error-correcting code duality under FHE.** We explicitly state the connection between compression and error-correcting codes. This motivates the research of the low-rate regime. We hope that the motivation we give in this paper will motivate information theorists to research this regime.
- **Novel compression algorithms under FHE.** We give 2 novel FHE-friendly compression algorithms that are inspired by ECC. One is inspired by LDPC the other is inspired by linear codes.
- We provide **implementations** for our algorithms implemented with HElayers[1].

We summarize our algorithms and compare them to previous work in Table 1.

	Comp. Time	Success Prob.	Comm.
Compression and Decompression Under FHE			
Copy&Recurse (Alg. 1,2)	$O(s^3 \log \frac{n}{s} + ns)$	1	$O(s \log \frac{n}{s})$
Only Compression Under FHE			
Coding (Alg. 3)	$O(ns)$	< 1	$O(s)$
Peeling (Alg. 5)	$O(ns \log s)$	< 1	$O(s \log s)$
[4]	$O(s^2 n \log n)$	1	$O(s^2 \log n)$
[8](**)	$O(ns)$	< 1	$O(s)$
[13](**)	$O(n)$	< 1	$O(s)$
[16](**)	$O(n \log n)$	1	$O(s)$
[16](**)	$O(n)$	< 1	$O(s)$

Table 1: Summarizing and comparing our algorithms and previous work. The first column shows the time (circuit size) needed to compute the compression. The second column shows the success probability of the decompression. The third column shows the communication (compressed) size. Here n, s are the size and sparsity of the input. A success probability of < 1 means it can be arbitrarily close to 1 (while increasing the communication and compression time). (*) With copy-and-recurse, the decompression (Algorithm 2) can be executed under FHE with the same circuit bounds.
(**) This is for the batch PIR problem.

2 Preliminaries and Related Work

2.1 Related Work

In cleartext, the trivial solution to compressing a s-sparse vector (returning the values and indexes of nonzero entries) is optimal, and as such it was not studied. Under FHE this problem is significantly harder and below we review the previous works.

Under FHE, optimal communication (ignoring factors depending on security parameter) can be achieved by iteratively using the SPiRiT sketch of [3] that extracts the first nonzero element, and then resetting this element. The sketch uses $isZero(x) = \begin{cases} 1, & x = 0, \\ 0, & x \neq 0. \end{cases}$ which they showed how to implement efficiently under BGV by performing the computation using $O(\log n)$ keys with different plaintext moduli. Reconstructing the index of the first nonzero element is then done in cleartext. Iteratively using SPiRiT would therefore require an interactive protocol with s rounds. In a parallel work [4], the same authors introduced a different approach, where the server uses additive homomorphic encryption to produce a coreset of the nonzero elements. The authors used ideas from *group testing* to reconstruct the indexes of the nonzero elements. In addition, they showed the problem of compressing a sparse vector $\mathbf{v}$ can be reduced to encoding the indexes of the nonzero elements.

[13] continued the work of [4] and considered the problem of retrieving the indexes of the nonzero elements in $\mathbf{v}$. They also considered the problem of retrieving the nonzero values in $\mathbf{v}$ but they used a tailored solution, unlike the generic reduction shown in [4]. They showed 2 solutions to fetch the indexes. They first used arithmetic bloom filters. This is a special case of a linear code with a binary generating matrix. Their second solution is also a special case of a linear code but in this case the generating matrix is a Vandermonde matrix.

In [16] the authors gave 2 solutions to the compression problem. The first solution defines a polynomial with the input being the coefficients, $P(x) = \prod v_i x^i$. Then they use FFT to efficiently encode $P(x)$. From the perspective of error correction, this approach can be interpreted as a Reed–Solomon code. The second algorithm uses inverted Bloom lookup tables (IBLT). The decryption then iteratively finds an element in the IBLT, by finding cells with a single element mapped to them.

Some of the previous works consider a related problem called *batch PIR* so before describing them we explain this problem and its relation to compressing s-sparse vectors.

In *batch PIR* we are given a vector $\mathbf{v} \in \mathbb{F}^n$ and a binary vector $\chi \in \{0,1\}^n$ that represents a set of indexes $I = \{i_1, i_2, \ldots\}$, i.e. $\chi_j = 1$ iff $j \in I$. The goal is to compute a small representation $batchPIR(\mathbf{v}, \chi)$ from which the values $v_{i_1}, v_{i_2}, \ldots$ can be reconstructed. Here, $\mathbf{v}$ itself cannot be reconstructed because the indexes of these values are not reconstructed. Compressing a s-sparse vector can be reduced to $batchPIR$ as we show next. First, recall that [4] showed that given a sparse (non binary) vector $\mathbf{v} \in \mathbb{F}^n$ it can be decomposed to $O(\log |\mathbb{F}|)$ sparse binary vectors. It remains to show how a sparse binary vector $\mathbf{v} \in \{0,1\}^n$ can be compressed using $batchPIR$. This is easy since it is trivial to reconstruct $\mathbf{v}$ from $batchPIR(\mathbf{l}, \mathbf{v})$, where $\mathbf{l} = (1, 2, \ldots)$ is the label vector.

The problem of batch PIR was recently studied by [8] who improved the work of [14] that used *sparse random linear coding* [26]. They also considered the inverse problem where the client has a list of values $(v_1, v_2, \ldots)$ and a list of indexes $I = (i_1, \ldots, i_s)$ and the goal is to privately transfer to the server a vector v' such that $v'_i = v_i$ if $i \in I$ (there is no

guarantee for the value v'_i otherwise). Privacy here means the server does not learn any value in v'. They referred to this problem as decompression although v' is not a reconstruction of v. In their experiments, they tested with s-sparse vectors of size n, where $s \approx n$. This is different from the asymptotic settings we consider. The compression algorithm they describe is a special case of a linear-code and the matching decompression algorithm includes solving a linear equation system which is inefficient to run under FHE.

2.2 FHE

Our algorithms require only "addition" and "multiplication" and can be implemented using any MPC scheme that supports them. However, to improve readability we describe our protocol using FHE. The experiments in Section 6 are also done under FHE.

FHE (see e.g. [9, 10]) is an asymmetric encryption scheme that is also equipped with $+$ and $\times$ operations on ciphertexts. More specifically, a FHE scheme is the tuple $\mathcal{E} = (Gen, Enc, Dec, Add, Mult)$, where:

- $Gen(1^\lambda, \ldots)$ gets a security parameter λ and other scheme-specific parameters and generates the keys pk and sk.
- $Enc_{pk}(m)$ gets a message m and outputs a ciphertext $[\![m]\!]$.
- $Dec_{sk}([\![m]\!])$ gets a ciphertext $[\![m]\!]$ and outputs a message m'.
- $Add_{pk}([\![a]\!], [\![b]\!])$ gets two ciphertexts $[\![a]\!], [\![b]\!]$ and outputs a ciphertext $[\![c]\!]$.
- $Mult_{pk}([\![a]\!], [\![b]\!])$ gets two ciphertexts $[\![a]\!], [\![b]\!]$ and outputs a ciphertext $[\![d]\!]$.

Correctness is the requirement that $m = m'$, $Dec([\![c]\!]) = a+b$ and $Dec([\![d]\!]) = a \cdot b$. In some FHE schemes (e.g. BGV [9]) the operations are done over $\mathbb{Z}_p$, where $p \in \mathbb{Z}$ is a parameter of the key which is set during key generation $Gen(1^\lambda, p, \ldots)$. In an approximated FHE (e.g. CKKS [10]) we require that $m \approx m'$, $Dec([\![c]\!]) \approx a + b$ and $Dec([\![d]\!]) \approx a \cdot b$.

Semantic security is the requirement that given $pk, [\![m_0]\!], [\![m_1]\!], \cdots [\![m_{poly(\lambda)}]\!], m_1, \ldots, m_{poly(\lambda)}$, where $poly(\lambda)$ is a number that is polynomial in λ, the value of m_0 is known with probability negligible in λ.

Using additions and multiplications we can compute any *arithmetic circuit* with encrypted inputs. For example, in a client-server system, the client encrypts her data and sends it to the server who computes the output of the circuit on the encrypted input. The output is also encrypted and is returned to the client who then decrypts it. The semantic security of FHE guarantees the server does not learn anything on the content of the client's data.

When evaluating an arithmetic circuit, C, we are concerned with the size of C, (denoted $\mathsf{Size}(C)$) which is the number of operations in C. Sometimes we are also interested in the multiplicative depth of C, (denoted $\mathrm{Depth}(C)$) which is the maximal number of multiplication gates on a path in C.

Abbreviated syntax. To make our algorithms more intuitive to read we use $[\![\cdot]\!]_{pk}$ to denote a ciphertext. When pk is clear from the context we omit it. We use an abbreviated syntax:

- $[\![a]\!] + [\![b]\!]$ is short for $Add_{pk}([\![a]\!], [\![b]\!])$.
- $[\![a]\!] \cdot [\![b]\!]$ is short for $Mult_{pk}([\![a]\!], [\![b]\!])$.
- $[\![a]\!] + b$ is short for $Add_{pk}([\![a]\!], Enc_{pk}(b))$.
- $[\![a]\!] \cdot b$ is short for $Mult_{pk}([\![a]\!], Enc_{pk}(b))$.

Unique Challenge of FHE: No Branching. The most challenging aspect of FHE is not being able to choose one of two execution paths (to branch), based on encrypted input. A general recipe is to use multiplexing. E.g., the cleartext code:

```
if x < y:
   min := x
else:
   min := y
```

is replaced with:

$$[\![min]\!] := [\![\chi_{\{x<y\}}]\!] \cdot [\![x]\!] + (1 - [\![\chi_{\{x<y\}}]\!]) \cdot [\![y]\!],$$

where $\chi_{\{x<y\}} = 1$ if $x < y$ and $\chi_{\{x<y\}} = 0$ otherwise (see for example [11] on how to compute $\chi_{\{x<y\}}$ under FHE). Unfortunately, using the generic recipe can be extremely inefficient. For example, traversing a binary search tree in cleartext translates to visiting every node of the tree under FHE. In our case, the folklore algorithm to compress a s-sparse vector generates an output of size s as follows: it keeps a pointer p, initially set to 1 and scans the input vector. If the scan reaches a nonzero value, its index is written to the pth place in the output and p is incremented by 1 (see Appendix C for implementation details).

Ring Size. An interesting difference between cleartext and FHE is the unit of communication. In cleartext the communication unit is a single bit and communicating a ring element $a \in \mathbb{F}$ requires communicating $\lceil \log_2 |\mathbb{F}| \rceil$ bits. In FHE, on the other hand, the communication unit is an encrypted message. Here the actual communication size depends on the ring size but also on the security parameter of the encryption. More specifically, the communication size grows sub-logarithmically with the ring size.

2.3 Tree Notation

Some data structures we use are based on trees. When describing a tree we use v to denote a node and use dot (".") to denote members of v. For example, $v.child[1], \ldots, v.child[r]$ are the children of v. The height of a node v is the number of nodes on the path from v to the root and the height of a tree is the maximal height of its nodes.

2.4 Copy-And-Recurse

Copy-and-recurse is a technique from [22] (later improved in [23]) to traverse a tree efficiently under FHE. With copy-and-recurse, a cleartext algorithm can be converted to run under FHE by paying an overhead linear in the tree size. In more detail, to use copy-and-recurse on a tree T with n leaves it is required that: (1) every inner node of T has r children; and (2) every leaf of T is at the same distance from the root.

Suppose there is a cleartext algorithm $\mathcal{A}$ that traverses a tree such that (1) at each level, $\mathcal{A}$ visits at most $c \geq 1$ children (where c is a constant); and (2) at every node v that $\mathcal{A}$ visits it computes a function $f(v)$ which takes t time to compute. Then $\mathcal{A}$ takes $O(t \log n)$ time to run in cleartext and the naïve FHE implementation would take $O(tn)$ time to run. Using copy-and-recurse, $\mathcal{A}$ can be implemented under FHE in time $O(t \log n + n)$. Note that at each node v, $\mathcal{A}$ needs to decide which children it should continue its traversal in. This usually involves comparisons that are negligible in cleartext but are time-costly under FHE. Here, the costs of the comparisons are counted in the term t, and using copy-and-recurse the total cost of all comparisons drops from $O(tn)$ to $O(t \log n)$.

The main idea behind copy-and-recurse is to: (1) determine (under FHE) at each level which c of the $c \cdot r$ children the cleartext algorithm would have continued into; (2) generate a selection matrix, S (see Section 2.5 for details on selection matrices), that "selects" those c children; (3) multiply S by the vector of children to copy the selected children and their subtrees (again under FHE); and then (4) recurse into those c copies. The trick here is that although copying the c children takes $O(n)$ time under FHE, after doing so, the number of times we compute $f(\cdot)$ is similar to that of the cleartext version. See Figure 3 for a depiction.

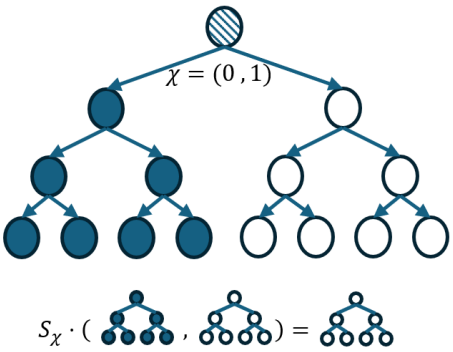

Figure 3: A depiction of copy-and-recurse. The top shows a binary search tree. At the root, the cleartext algorithm $\mathcal{A}$ decides to go right. The equivalent FHE algorithm computes an indicator vector $\chi = (0, 1)$ indicating it should continue to the right child. The left and right subtrees are multiplied by a selection matrix, whose output is the right subtree. The process is then repeated. In this case, although there is an $O(n)$ overhead, the expensive comparison operation is called only $O(\log n)$ times (to compute the indicator vector).

2.5 Selection And Expansion Matrix

To copy multiple subtrees copy-and-recurse uses a *selection matrix* which we now describe in more detail. Given a d-dimensional binary vector (often called an *indicator vector*) with Hamming weight $c < d$ (i.e. c elements are nonzero). A selection matrix $S(\chi) \in \{0,1\}^{c \times d}$ is a matrix that has the property that it *selects* elements indicated by χ. That is, if $i_1, \ldots, i_c$ are the indexes of the nonzero elements in χ then for any vector $x = (x_1, \ldots, x_d)$ we have $S \cdot x^T = (x_{i_1}, \ldots, x_{i_c})$. Intuitively, a selection matrix is a generalization of the *multiplexer* (MUX) operation. To construct S we set $S_{a,b} = 1$ if the b-th element in x is the a-th nonzero element, otherwise we set $S_{a,b} = 0$. Algorithm 1 in [22] shows how to construct a selection matrix using a circuit of size $O(c \cdot d^2)$ and depth $O(c \cdot \log d)$.

Given a selection matrix S as above, its matching expansion matrix $Z(\chi) = S(\chi)^T$ has the property that if $x' = Z \cdot (S \cdot x)$ then $x'_k = x_j$ if $k = i_j \in \{i_1, \ldots, i_c\}$ and $x'_k = 0$ otherwise.

2.6 Error Correcting Codes

Codes and linear codes An *error correcting code* is a mapping $\mathcal{E} : \mathcal{M} \mapsto \mathcal{X}^n$, where $\mathcal{M}$ is a discrete set of messages, $\mathcal{X}$ is an alphabet at the input of a given communication channel, and n is the number of times the channel can be used. The members of $\mathcal{M}$ are called *codewords*. The size of the code is $|\mathcal{M}|$, the length of the code is n, and the rate of the code is $R = \frac{1}{n} \log_2 |\mathcal{M}|$, namely the average number of message bits per channel use. Often times, one takes $\mathcal{M} = \{0,1\}^k$, and then the rate is $R = k/n$. The communication channel maps input vectors $X^n \in \mathcal{X}^n$ to output vectors $Y^n \in \mathcal{Y}^n$, where $\mathcal{Y}$ is the channel output alphabet. This mapping can be either stochastic or adversarial (unknown deterministic under some constraints), depending on the setting. A decoding algorithm is a mapping $\mathcal{D} : \mathcal{Y}^n \mapsto \mathcal{M}$, mapping channel outputs to reconstructed messages. Typically, we want to be able to reconstruct the correct message, i.e., have $\mathcal{E}(M) = \mathcal{D}(Y^n)$ either with high probability (stochastic setting) or for any possible channel realization (adversarial setting). Often times, an optimal decoder for a given code can be inefficient, and we may opt for a suboptimal yet efficient one.

The basic stochastic setting is that of a discrete memoryless channel, where the channel maps each input independently to its corresponding output according to a fixed stochastic law. Shannon's channel coding theorem [24] exactly characterizes the channel capacity, namely the maximal rate R of a code that can be decoded with vanishing error probability in the limit of large n. Efficient codes that work well empirically in specific channels, such as Turbo Codes [7] and Low Density Parity Codes (LDPC) [17], have been known for a long time. The first efficient constructions that provably achieve capacity for a wide family of channels, known as Polar Codes, were introduced more recently [5]. The basic adversarial settings is where the channel can insert up to a fraction p of input errors. The maximal rate of a code that can correct up to a fraction p of errors is a fundamental open question in coding theory, with various known upper and lower bounds. One well-known and very useful construction is the Reed-Solomon (RS) code, based on polynomial interpolation. The RS code can efficiently achieve a rate of $\approx (1-2p) \log_2(|\mathcal{X}|)$, whenever $\mathcal{X}$ is a large enough finite field.

Linear codes are a special family of error correcting codes, where encoding is a linear operation between vector spaces over the same field, or equivalently, where the codewords form a linear subspace. Namely, in this case $\mathcal{M} = \mathbb{F}^k$, $\mathcal{X}^n = \mathbb{F}^n$, and $\mathcal{E}(X^k) = G \cdot X^k$, where $G \in \mathbb{F}^{n \times k}$ is the so-called generator matrix of the code. Linear codes are often equivalently characterized by their parity check matrix $H \in \mathbb{F}^{(n-k)\times n}$, which spans the kernel of the codewords subspace, i.e., satisfies $Hc = 0$ if and only if c is a codeword. Decoding algorithms for linear codes are non-linear, but linearity makes it easier to find efficient ones. Most of the codes suggested in the literature and used in practice are linear, including Turbo codes, LDPC, Polar codes and RS codes.

Compression is the task of mapping an input vector $\mathcal{A} \in \mathbb{F}^n$ into a shorter vector $\mathcal{C} \in \mathbb{F}^k$ ($k < n$), such that $\mathcal{A}$ can be reconstructed from $\mathcal{C}$. The associated compression rate is then $R = k/n$. Obviously, this is impossible in the worst case, when all $|\mathbb{F}|^n$ possibilities of $\mathcal{A}$ need to be reconstructed. Thus, the question is typically considered in one of two ways: 1) under a stochastic assumption on the data generation source, in which case a small error probability is allowed, or alternatively, a variable length code is allowed; 2) assuming the data is supported on a small set, e.g., sparse vectors. The simplest stochastic setting is the discrete memoryless source, where each entry of the vector $\mathcal{A}$ is drawn independently from the same distribution. In this case, the Shannon source coding theorem [24] shows that the best possible compression rate is the entropy of the information source. Shannon's optimal construction is asymptotic and non-efficient, but there are explicit and often efficient constructions, e.g. the Huffman code [21], and arithmetic coding [27] which can be generalized far beyond the i.i.d. case. Going even further, Lempel and Ziv [28] showed how to efficiently and universally compress and decompress information sources with unknown distributions, or even ones without any stochastic assumptions, asymptotically achieving the so-called finite-state entropy of the underlying model or specific individual sequence. In this paper, we are interested in compressing s-sparse vectors, i.e., vectors for which at most s elements are nonzero. When there are no constraints on the compression algorithm, this problem is typically trivial. However, under the FHE paradigm, we will see it becomes quite challenging.

Duality of compression and channel-coding. Error correcting codes can sometimes be used to obtain compression schemes and vice versa, by essentially reversing the roles of the encoder and the decoder. This duality is sharp in the case of lossless compression using linear codes (and slightly different in other cases, see [20]). The idea is the following. Suppose a linear code over (say) $\mathbb{F}_2$ with generator G and parameters (n, k), hence rate k/n, can correct up to a fixed fraction p of random (resp. adversarial) errors with low (resp. zero) error probability. This essentially means that for n-vectors w that are iid $Ber(p)$ (resp. of Hamming weight $\leq np$), there is a mapping $Hw \mapsto w$ that is correct with high probability (resp. always correct), where H is the parity check matrix. To see this, we note that $H(Gx + w) = Hw$, and so such a mapping must exist. Therefore, we can losslesly compress such vectors w by simply using Hw as the encoding, which has $n - k$ bits, hence overall attaining a compression rate of $1 - k/n$. This conversion clearly works also in the other direction (from compression to error correcting codes), and is tight, namely translating optimal-rate channel codes to optimal-rate compression schemes.

Interestingly, there is a more general way to obtain (possibly efficient) error correcting codes from (not necessarily linear) efficient compression schemes. To keep things simple, suppose our data is in $\mathcal{S} \subseteq \mathbb{F}^n$ (e.g., s-sparse vectors), and we have a compression algorithm with efficient encoding $\mathcal{S} \to \mathbb{F}^k$ and efficient decoding (without the efficiency constraints, this is trivial by simply enumerating $\mathcal{S}$). While the encoding algorithm is defined only on $\mathcal{S}$, often it can be naturally extended to the entire $\mathbb{F}^n$, as we now assume. A vector $w \in \mathbb{F}^n$ is called *invariant* (w.r.t. to the encoding and $\mathcal{S}$) if, for any given $x \in \mathcal{S}$, the vectors x and $x + w$ have the same encoding. Let $\mathcal{I} \subseteq \mathbb{F}^n$ be a set of invariant vectors. Then $\mathcal{I}$ is an error correcting code that can correct any additive error $x \in \mathcal{S}$, with (essentially) an efficient decoding algorithm: simply apply the compression scheme encoder, which by invariance will be the compressed version of the error vector x only, then apply the compression scheme decoder to recover the error vector x, and finally subtract the error to recover the input codeword $w \in \mathcal{I}$.

Our focus. In this paper, we focus on the compression problem in the regime where $s \ll n$ (e.g., s is fixed or logarithmic in n) together with the constraint of linear encoding, or more generally, encoding computed by evaluating a low degree polynomial. This is important for attaining good runtime in the context of homomorphic evaluation. In this small-s regime, such schemes are not interesting for cleartext encoding, since we can trivially (non-linearly) efficiently compress using $s \log n$ bits, which is essentially optimal for large n. However, when data is given in encrypted form, such a simple scheme may not be applicable, and constraints on the encoding process (such as linearity or low-degree) render the compression problem more complex and interesting. From a linear coding perspective, which we take often in this work, we are thus interested in codes that correct a small number of errors s, and have a rate approaching 1.

3 Problem Statement

A *compression* algorithm is an algorithm that takes a vector $\mathbf{v} \in S \subset \mathbb{F}^n$, where $|S| \leq |\mathbb{F}|^m$ and outputs a vector $\mathbf{v}' \in \mathbb{F}^m$, such that there is an inverse algorithm called *decompression* that gets $\mathbf{v}'$ and outputs $\mathbf{v}$. Specifically, we are interested in the case where S is the set of all s-sparse vectors. In the secure version of the problem, the input to the compression, $\mathbf{v}$, is encrypted and the input to the decompression, $\mathbf{v}'$, is also encrypted.

An example of motivation is described in Figure 1 where a client sends an encrypted query to one or more servers; the servers reply to an untrusted agent service that aggregates

and analyses the replies and then returns the final result to the client.

The parties we consider are semi-honest and computationally bounded. The server and the agent do not learn anything on the content of the encrypted messages in their view and the client does not learn anything but the reply to their query.

4 Compression And Decompression Under FHE

In this section we describe our FHE friendly compression and decompression algorithms. Our compression algorithm compresses a s-sparse binary vector $\mathbf{a}$, of size n using a circuit of size $O(s^3 \log \frac{n}{s} + ns)$ and depth $O(s \log \frac{n}{s} \log s)$, The size of the compressed vector is $O(s \log \frac{n}{s})$ ciphertexts. The decompressor can then reconstruct $\mathbf{a}$ with a circuit of the same size and depth.

Before describing our algorithms we describe a binary tree, T, built for $\mathbf{a}$. It is easier to understand the description and intuition of our algorithms with this tree in mind.

We construct a balanced binary tree, T, with n leaves. For simplicity, assume that n is a power of 2, T is full and has $\log_2 n$ levels, level 0 being the root and level $\log_2 n$ being the leaves. Denote by $x_1^{(\ell)}, \ldots, x_{2^\ell}^{(\ell)}$ the nodes in level ℓ of the tree. We associate the leaf $x_i^{(\log n)}$ with the value $\mathbf{a}[i]$ and associate an inner node $x_i^{(\ell)}$ with the value 1 if at least one leaf in its subtree is nonzero. Abusing the notation, $x_i^{(\ell)}$ also denote these values. See Figure 4 for an example of a tree built for a 4-sparse vector of size 16.

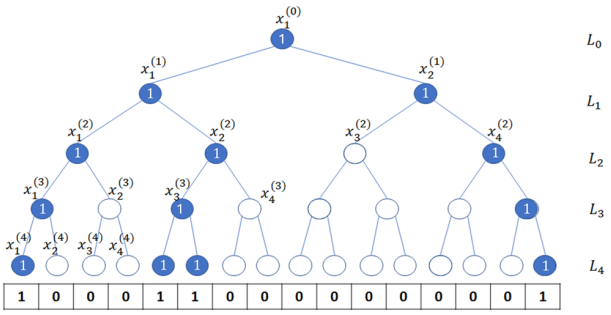

Figure 4: A vector with $n = 16$ cells and sparsity of $s = 4$ (shown at the bottom) and the binary tree we associate with it (above it). The i-th node in the ℓ-th level is denoted as $x_i^{(\ell)}$. In each node we keep an indicator whether at least one of the cells associated with leaves in its subtree is nonzero. For clarity, the nodes with nonzero values are colored and the value is written inside them.

Observation 1. *We observe these on T:*

(1) *If $x_i^{(\ell)} = 0$ then its children $x_{2i}^{(\ell+1)} = x_{2i+1}^{(\ell+1)} = 0$.*

(2) *At each level ℓ we have $\sum_i x_i^{(\ell)} \leq s$.*

(3) *If $x_i^{(\ell)}$ is nonzero, then at least one of its children $x_{2i}^{(\ell+1)}, x_{2i+1}^{(\ell+1)}$ is nonzero.*

Algorithm 1: *copyRecurseComp*$_{n,s}(\llbracket A \rrbracket)$

Parameters: n, s, where n, s are powers of 2.
Input: A s-sparse vector $\llbracket \mathbf{a} \rrbracket = (\llbracket \mathbf{a}[1] \rrbracket, \ldots, \llbracket \mathbf{a}[n] \rrbracket) \in \{0,1\}^n$.
Output: $\llbracket \mathbf{a}[i_1] \rrbracket, \ldots, \llbracket \mathbf{a}[i_s] \rrbracket \in \{0,1\}$ and $\llbracket \chi_1 \rrbracket, \ldots, \llbracket \chi_{(\log \frac{n}{s})} \rrbracket \in \{0,1\}^{2s}$.

1 **foreach** $\alpha = 1, \ldots, n$ **do**
2 $\quad \llbracket x_\alpha^{(\log n)} \rrbracket := \llbracket \mathbf{a}[\alpha] \rrbracket$
3 **foreach** $\ell = \log n - 1, \ldots, \log s$ **do**
4 $\quad$ **foreach** $\alpha = 1, \ldots, 2^\ell$ **do**
5 $\quad\quad \llbracket x_\alpha^{(\ell)} \rrbracket := \llbracket x_{2\alpha-1}^{(\ell+1)} \rrbracket + \llbracket x_{2\alpha}^{(\ell+1)} \rrbracket - \llbracket x_{2\alpha-1}^{(\ell+1)} \rrbracket \cdot \llbracket x_{2\alpha}^{(\ell+1)} \rrbracket$
6 Denote by $T^{(\ell)} = (T_1^\ell, \ldots, T_s^\ell)$ a vector of subtrees.
7 $\llbracket T_i^{(\log s)} \rrbracket :=$ the sub-tree under $x_i^{(\log s)}$ for $i = 1, \ldots, s$
8 **foreach** $\ell = \log s, \ldots, \log n - 1$ **do**
9 $\quad \llbracket X^{(\ell)} \rrbracket := (\llbracket T_1^{(\ell)}.left \rrbracket, \llbracket T_1^{(\ell)}.right \rrbracket, \ldots, \llbracket T_s^{(\ell)}.left \rrbracket, \llbracket T_s^{(\ell)}.right \rrbracket)$
10 $\quad \chi_i^{(\ell)} :=$ the value at the root of $X_i^{(\ell)}$, for $i = 1, \ldots, 2s$
11 $\quad \llbracket S^{(\ell)} \rrbracket := BuildSelectionMatrix_{2s,s}(\llbracket \chi^{(\ell)} \rrbracket)$
12 $\quad \llbracket T^{(\ell+1)} \rrbracket := \llbracket S^{(\ell)} \rrbracket \cdot \llbracket X^{(\ell)} \rrbracket$
13 **Output** $\llbracket \chi^{(\log s)} \rrbracket, \ldots, \llbracket \chi^{(\log n - 1)} \rrbracket$

Overview of copyRecurseComp. Algorithm 1 has 2 parameters $0 < s < n$, which for simplicity, assume are powers of 2. It also gets as input a s-sparse vector $\mathbf{a}$ of size n. The algorithm constructs a binary tree T (Lines 2-5) as described above. Since it does not use the top-most $\log s$ levels, their construction is skipped.

Next, Algorithm 1 uses c-level copy-and-recurse to traverse the tree from top to bottom. During the traversal, it maintains the subtrees that have nonzero leaves. Since $\mathbf{a}$ is s-sparse at any level, ℓ, there are at most s such subtrees, denoted $T_1^{(\ell)}, \ldots, T_s^{(\ell)}$.

The traversal starts at level $\ell = \log s$ where $T_1^{(\log s)}, \ldots, T_s^{(\log s)}$ are trivially set to be all s subtrees in that level (Line 7). For the next levels it computes $T_1^{(\ell+1)}, \ldots, T_s^{(\ell+1)}$ by copying s out of the $2s$ children of $T_1^{(\ell)}, \ldots, T_s^{(\ell)}$ as follows. It sets $X^{(\ell)}$ to be a vector of $2s$ subtrees which are the children, $T_i^{(\ell)}.left$ and $T_i^{(\ell)}.right$, for $i = 1, \ldots, s$ (Line 9) (we remind that although the content of each $T_i^{(\ell)}$ is encrypted, the structure of the tree is known). Next, Algorithm 1 sets $\chi^{(\ell)}$ to be the (encrypted) values at the roots of $X_1^{(\ell)}, \ldots, X_{2s}^{(\ell)}$ (Line 10). This serves as the indicator vector, indicating which subtrees we need to recurse into. Then, the algorithm constructs the selection matrix $M^{(\ell)}$ from $\chi^{(\ell)}$ (Line 11) and uses it to make a copy of the s nonzero subtrees $T_1^{(\ell+1)}, \ldots, T_s^{(\ell+1)}$ (Line 12). The algorithm stops when it reaches the leaves.

The output of the compression algorithm is $\chi^{(\log s)},\ldots,\chi^{(\log n-1)}$, where each $\chi^{(\ell)}$ is an indicator vector of size $2s$.

THEOREM 4.1. *Algorithm 1 can be computed with a circuit of size $O(s^3\log\frac{n}{s}+ns)$ and depth $O(s\log\frac{n}{s}\log s)$.*

For lack of space, we move the proof to Appendix A.1

Algorithm 2: $copyRecurseDecomp_{n,s}(\llbracket\chi^{(\log s)}\rrbracket,\ldots,\llbracket\chi^{(\log n-1)}\rrbracket)$

Parameters: Same parameters as Algorithm 1
Input: $\chi^{(\log s)},\ldots,\chi^{(\log n-1)}$ (output of Algorithm 1)
Output: $\llbracket\mathbf{a}\rrbracket$, where $\mathbf{a}$ was the input to Algorithm 1

1 **foreach** $i=1,\ldots,s$ **do**
2 $\quad\llbracket T_i^{(\log n-1)}.left\rrbracket := \llbracket\chi_{2i-1}^{(\log n-1)}\rrbracket$
3 $\quad\llbracket T_i^{(\log n-1)}.right\rrbracket := \llbracket\chi_{2i}^{(\log n-1)}\rrbracket$
4 **foreach** $\ell=\log n-2,\ldots,\log s$ **do**
5 $\quad\llbracket Z^{(\ell)}\rrbracket := BuildExpansionMatrix_{2s,s}(\llbracket\chi^{(\ell)}\rrbracket)$
6 $\quad\llbracket X^{(\ell)}\rrbracket := \llbracket Z_\ell\rrbracket\cdot\llbracket T^{(\ell+1)}\rrbracket$
7 $\quad$**foreach** $i=1,\ldots,s$ **do**
8 $\quad\quad\llbracket T_i^{(\ell)}.left\rrbracket := \llbracket X_{2i-1}^{(\ell)}\rrbracket$
9 $\quad\quad\llbracket T_i^{(\ell)}.right\rrbracket := \llbracket X_{2i}^{(\ell)}\rrbracket$
10 $\llbracket\mathbf{a}\rrbracket :=$ the leaves in $\llbracket T_1^{(\log s)}\rrbracket,\ldots,\llbracket T_s^{(\log s)}\rrbracket$
11 **Output** $\llbracket\mathbf{a}\rrbracket$

Overview of copyRecurseDecomp. Algorithm 2 is the inverse of Algorithm 1 in the sense that $Decomp(Comp(\llbracket\mathbf{a}\rrbracket))=\llbracket\mathbf{a}\rrbracket$ when $\mathbf{a}$ is s-sparse (the equality here means that the messages in the ciphertexts are equal). The parameters of Algorithm 2 are n,s, the same as Algorithm 1, where we assume for simplicity that they are powers of 2. Its input is the output of Algorithm 1: $\llbracket\chi^{(\log s)}\rrbracket,\ldots,\llbracket\chi^{(\log n-1)}\rrbracket$.

The algorithm starts by constructing $\llbracket T_1^{(\log n-1)}\rrbracket,\ldots,\llbracket T_s^{(\log n-1)}\rrbracket$ by setting their left and right children to the encrypted values in $\llbracket\chi^{(\log n-1)}\rrbracket$ (Lines 2-3). Then, it iteratively builds (as we explain below) the trees in $T_1^{(\ell)},\ldots,T_s^{(\ell)}$ for $\ell=\log n-2,\ldots,s$. Finally, having constructed $\llbracket T_1^{(\log s)}\rrbracket,\ldots,\llbracket T_s^{(\log s)}\rrbracket$ gives all the leaves of T.

To construct the trees of level ℓ from those of level $\ell+1$, Algorithm 2 first computes the expanding matrix $Z^{(\ell)}$ from $\chi^{(\ell)}$ (Line 5). Then it calculates the product $X^{(\ell)}=\llbracket Z^{(\ell)}\rrbracket\cdot\llbracket T^{(\ell+1)}\rrbracket$ (Line 6). Here, $X^{(\ell)}$ and $T^{(\ell)}$ are vectors whose elements are subtrees. A zero element (which the expansion matrix constructs) in this case is a tree whose nodes are all zeros. Next, it connects $X_{2i-1}^{(\ell)}$ and $X_{2i}^{(\ell)}$ by setting them to be the left and right children of a single tree which is set to be $T_i^{(\ell)}$ (Lines 8,9).

The traversal ends after the algorithm constructs $T_1^{(\log s)},\ldots,T_s^{(\log s)}$. Then all the leaves of the tree have been constructed and the output is set to their values (Line 11).

THEOREM 4.2. *Algorithm 2 can be computed with a circuit of size $O(s^3\log\frac{n}{s}+ns)$ and depth $O(s\log\frac{n}{s}\log s)$.*

For lack of space we move the proof to Appendix A.2

5 Compression Using Error-Correcting Codes Under FHE

In this section we continue the work of Section 4 and show two novel FHE friendly compression algorithms that are based on ECC.

5.1 A Tree-Based Code with FHE-Friendly Compression

We now describe compression and decompression algorithms that are based on a linear ECC and take advantage of the sparsity of the input. Here, the compression is FHE friendly and can be computed under FHE with a shallow and small circuit. The decompression, on the other hand, requires solving a linear equation system for which no efficient algorithm is known under FHE.

In more detail, our compression algorithm encodes a s-sparse binary vector $\mathbf{a}$, of size n using a circuit of depth $O(1)$ and size $O(m(n-m))$, where $s\le m\le n$ is a parameter that trades success probability with a smaller output: the output size of the compression is $O(m\log\frac{n}{m})$ ciphertexts and the decompressor can then reconstruct $\mathbf{a}$ (in plaintext) in time $O(s^3\log\frac{n}{m})$ with probability $O(|\mathbb{F}|^{(2s-m-1)(\log\frac{n}{m})})$.

Algorithm 3: $treeComp_{n,s,m}(\llbracket\mathbf{a}[1]\rrbracket,\ldots,\llbracket\mathbf{a}[n]\rrbracket)$

Parameters: A field $\mathbb{F}$, $0<s$ and $2s\le m<n<|\mathbb{F}|$. For simplicity, assume m,n are powers of 2
Input: An encrypted s-sparse vector $(\llbracket\mathbf{a}[1]\rrbracket,\ldots,\llbracket\mathbf{a}[n]\rrbracket)$, s.t. $\mathbf{a}[i]\in\{0,1\}$.
Output: A code $\mathbf{b}$, with $|\mathbf{b}|=O(m\log n)$, such that $\mathbf{a}$ can be reconstructed from $\mathbf{b}$ with probability $O(|F|^{(2s-m-1)(\log n-\log m}$).

1 **for each** $i=1,2,\ldots,n$ **do**
2 $\quad\llbracket x_i^{(\log n)}\rrbracket := \llbracket\mathbf{a}[i]\rrbracket$
3 **for each** $\ell=\log n-1,\ldots,\log m$ **do**
4 $\quad$**for each** $i=1,2,\ldots,2^\ell$ **do**
5 $\quad\quad\llbracket x_i^{(\ell)}\rrbracket := \llbracket x_{2i-1}^{(\ell+1)}\rrbracket+\llbracket x_{2i}^{(\ell+1)}\rrbracket$
6 $\llbracket\mathbf{y}^{(\log m)}\rrbracket := (\llbracket x_1^{(\log m)}\rrbracket,\ldots,\llbracket x_m^{(\log m)}\rrbracket)$
7 **for each** $\ell=\log m+1,\ldots,\log n$ **do**
8 $\quad S^{(\ell)}\leftarrow$ draw a random seed
9 $\quad$Generate a pseudo random matrix $M^{(\ell)}\in\mathbb{F}^{m\times 2^\ell}$ using $S^{(\ell)}$
10 $\quad\llbracket\mathbf{y}^{(\ell)}\rrbracket := M^{(\ell)}\cdot(\llbracket x_1^{(\ell)}\rrbracket,\ldots,\llbracket x_{2^\ell}^{(\ell)}\rrbracket)$
11 **Output:**
$S^{(\log m+1)},\ldots,S^{(\log n)},\llbracket\mathbf{y}^{(\log m)}\rrbracket,\ldots,\llbracket\mathbf{y}^{(\log n)}\rrbracket$

Overview of treeComp (Algorithm 3). The algorithm operates over a field $\mathbb{F}$, where $0<s\le m<n$ are public parameters. Also, we assume here $s<|\mathbb{F}|$. The input of the algorithm is a s-sparse binary vector $\mathbf{a}$ and the output of the algorithm is a code $\mathbf{c}$, where $|\mathbf{c}|=O(m\log\frac{n}{m})$, from which $\mathbf{a}$ can be reconstructed. The parameter m trades the output size with the success probability of the decompression.

We use the tree representation described in Section 4, with one difference that here each inner node keeps the sum of leaves in its subtree. The algorithm starts by building the tree. This is done in Lines 2- 5 where we calculate the values $x_i^{(\ell)}$ for $\log m \le \ell \le \log n$ and $1 \le i < 2^\ell$. Next, at each level $\log m + 1 \le \ell \le \log n$ it:

(1) draws a random seed $S^{(\ell)}$ and uses it to generate a pseudo random $m \times 2^\ell$ matrix $M^{(\ell)}$ (Line 8-9)
(2) calculates $\mathbf{y}^{(\ell)} = M^{(\ell)} \cdot (x_1^{(\ell)}, \ldots, x_{2^\ell}^{(\ell)})^T$ (Line 10)
(3) outputs $S^{(\ell)}$ and $\mathbf{y}^{(\ell)}$ (Line 11)

Algorithm 4: $treeDecomp_{n,s,m}(\{S^{(\ell)}, [\![\mathbf{y}^{(\ell)}]\!]\}_{\log m \le \ell \le \log n})$

Parameters: Same as Algorithm 3.
Input: $S^{(\log m+1)}, \ldots, S^{(\log n)}, [\![\mathbf{y}^{(\log m+1)}]\!], \ldots, [\![\mathbf{y}^{(\log n)}]\!]$.
Output: A, the decrypted input of Algorithm 3.

1 Decrypt $\mathbf{y}^{(\ell)} := \mathsf{Dec}([\![\mathbf{y}^{(\ell)}]\!])$, for all ℓ
2 $x_i^{(\log m)} = \mathbf{y}_i^{(\log m)}$, for $i = 1, \ldots, m$
3 **for each** $\ell = \log m + 1, \ldots, \log n$ **do**
4 Construct $M^{(\ell)}$ from $S^{(\ell)}$.
5 $C := (c_1, c_2, \ldots)$, for all c_i s.t., $x^{(\ell-1)}_{\lfloor \frac{c_i}{2} \rfloor} \neq 0$
  `/* Restrict` $M^{(\ell)}$ `and` $\mathbf{y}^{(\ell)}$ `to` $|C|$ `columns */`
6 $M'^{(\ell)} := (M^{(\ell)}[c_1], M^{(\ell)}[c_2], \ldots)$
7 $\mathbf{y}'^{(\ell)} := (\mathbf{y}^{(\ell)}[c_1], \mathbf{y}^{(\ell)}[c_2], \ldots)$
  `/* Restrict` $M'^{(\ell)}$ `to` $|C|$ `rows */`
8 $M''^{(\ell)} :=$ choose $|C|$ rows of $M'^{(\ell)}$ s.t. $M''^{(\ell)}$ is regular
9 $\mathbf{x}'^{(\ell)} = (M''^{(\ell)})^{-1} \cdot \mathbf{y}'^{(\ell)}$
  `/* Expand` $\mathbf{x}'^{(\ell)}$ `to` $\mathbf{x}^{(\ell)}$ `by adding 0s */`
10 $x_i^{(\ell)} = \begin{cases} x_k^{(\ell)}, & \text{if } c_k = i \\ 0, & \text{otherwise} \end{cases}$
11 **Output:** $x_1^{(\log n)}, \ldots, x_n^{(\log n)}$

Overview of treeDecomp (Algorithm 4). The input of Algorithm 4 is the output of Algorithm 3, i.e., $S^{(\log m+1)}, \ldots, S^{(\log n)}, [\![\mathbf{y}^{(\log m)}]\!], \ldots, [\![\mathbf{y}^{(\log n)}]\!]$. Specifically, when referring to the tree construction, T, these are: (1) $\mathbf{y}^{(\log m)}$ - the variables of level $\log m$, (2) the seeds used to generate the matrices and (3) the sketches $\mathbf{y}^{(\log \mathbf{m+1})}, \ldots \mathbf{y}^{(\log \mathbf{n})}$ that were calculated.

Algorithm 4 starts by decrypting the input sketches (Line 1) and level $\log m$ of T is taken from the input (Line 2). Then the algorithm iteratively reconstructs the next levels in Lines 3-10 as we now explain.

The algorithm calculates each level ℓ from the level preceding it by solving a linear equation system to find the unknowns: $x_1^{(\ell)}, \ldots, x_{2^\ell}^{(\ell)}$. We can reduce the number of unknowns in the system using Observation 1, that $x_i^{(\ell)} = 0$ if $x_{\lfloor i/2 \rfloor}^{(\ell-1)} = 0$ and remembering that there are at most s nonzero values in $x_1^{(\ell-1)}, \ldots, x_{2^{\ell-1}}^{(\ell-1)}$. It follows that we can eliminate (set to 0) all but $2s$ unknowns in the system.

In more detail, the algorithm uses the seed $S^{(\ell)}$ to construct the matrix $M^{(\ell)}$ (Line 4). Let C be the indexes of the unknowns that were not set to zero (Line 5) and use it to restrict $M^{(\ell)}$ and $\mathbf{y}^{(\ell)}$ to $|C|$ columns (Line 6). This leaves $|C| \le 2s \le m$ columns in $M'^{(\ell)}$. Next we remove rows from $M'^{(\ell)}$ to get a regular $|C| \times |C|$ matrix (Line 8). This succeeds with a probability that can be arbitrarily high (by increasing m). Finally, level ℓ is reconstructed by solving $x'^{(\ell)}$ (Line 9) and expanding it to $x^{(\ell)}$ (Line 10).

We repeat this process until we reach the last level $\log n$. At that point the leaves vold the value of the elements of $\mathbf{a}$.

A Few Notes.

(1) Algorithm 3 assumes $2s \le m$, this can be improved to $s \le m$. To do this, we add s rows to M' that correspond to the implicit equations $x_i^{(\ell-1)} = x_{2i-1}^{(\ell)} + x_{2i}^{(\ell)}$, for all i and ℓ.
(2) Reducing $M'^{(\ell)}$ to a regular $M''^{(\ell)}$ fails with some probability. This probability can be arbitrarily low by increasing m.
(3) Algorithm 3 requires that $n \le |\mathbb{F}|$ because at Line 5 it calculates:

$$[\![x_i^{(\ell)}]\!] := [\![x_{2i-1}^{(\ell+1)}]\!] + [\![x_{2i}^{(\ell+1)}]\!].$$

This can be changed to the non-linear assignment:

$$[\![x_i^{(\ell)}]\!] := [\![x_{2i-1}^{(\ell+1)}]\!] + [\![x_{2i}^{(\ell+1)}]\!] - [\![x_{2i-1}^{(\ell+1)}]\!] \cdot [\![x_{2i}^{(\ell+1)}]\!].$$

However, this means that the underlying code is no longer linear.
(4) Algorithm 3 constructs multiple matrices from multiple seeds. These can be united to generate a single large matrix that has a specific structure derived from our tree construction.
(5) Algorithm 3 assumes the values in $\mathbf{a}$ are binary. This can be extended to non-negative values.

Theorem 5.1. *Algorithm 3 can be implemented using a circuit with $2m(n-m)$ plaintext-multiplications. The size of its output, $[\![\mathbf{y}^{(\log m)}]\!], \ldots, [\![\mathbf{y}^{(\log n)}]\!]$ ciphertexts. Its output can be decompressed by Algorithm 4 with probability at least $|\mathbb{F}|^{(2s-m-1)\log \frac{n}{m}}$.* The proof is given in Appendix A.3.

5.1.1 Simplifying and connecting to ECC. At the heart of Algorithm 3 we compute $\mathbf{y}^{(\ell)} := M^{(\ell)} \cdot (x_1^{(\ell)}, \ldots, x_{2^\ell}^{(\ell)})$. Since each $x_i^{(\ell)}$ is linearly dependent on the elements of $\mathbf{a}$ we can appropriately replace $M^{(\ell)}$ to have $\mathbf{y}^{(\ell)} := M'^{(\ell)} \cdot \mathbf{a}$ and then concatenate all $M'^{(\ell)}$ to have $\mathbf{b} := M' \cdot \mathbf{a}$, where $\mathbf{b}$ is the concatenation of $\mathbf{y}^{(\log m)}, \ldots, \mathbf{y}^{(\log n)}$.

The kernel of this matrix $K = Kernel(M') \subset \mathbb{F}^n$ is an invariant set (in the sense discussed in Section 2.6) w.r.t. the encoding Algorithm 3 and the set of s-sparse vectors in $\mathbb{F}^n$. Therefore, K is a (linear) error correcting code of rate $\dim(K)/n$ that can correct any pattern of up to s errors, equipped with an efficient decoding algorithm comprised by applying Algorithms 3 and 4 to the channel output to recover the error vector (see Section 2.6).

5.2 A Peeling-Based Code with FHE-Friendly Compression

Our next compression solution is inspired by low-density parity-check (LDPC) codes. For intuition, consider a simpler (though less efficient) variation. Given a vector $\mathbf{a}$, the encoding consists of sums of random subsets of the elements of a. During decoding, we look for a *singleton*: a sum of elements in which all values are known except one. This allows us to deduce the missing element of $\mathbf{a}$. The algorithm follows this principle by generating sums of subsets of $\mathbf{a}$ and, during decompression, iteratively resolving singletons to recover the entire vector.

Algorithm 5: $peelComp_{n,s}([\![\mathbf{a}[1]]\!], \ldots, [\![\mathbf{a}[n]]\!])$

Parameters: n, s as in Algorithm 3. A generating matrix $G = G_{m\times n}$. A labeling matrix $L = L_{n\times n} \in \mathbb{Z}_n^{n\times n}$.
Input: $[\![\mathbf{a}]\!] = [\![\mathbf{a}[1]]\!], \ldots, [\![\mathbf{a}[n]]\!]$, where $\mathbf{a}[i] \in \{0, 1\}$.
1 $[\![\mathbf{g}]\!] := G \cdot [\![\mathbf{a}]\!]$
2 $[\![\ell]\!] := (G \cdot L) \cdot [\![\mathbf{a}]\!]$
3 **Output** $[\![\mathbf{g}]\!], [\![\ell]\!]$

Algorithm 6: $peelDecomp_{n,s}([\![\mathbf{g}]\!], [\![\ell]\!])$

Parameters: Same as of Algorithm 3.
Input: $[\![\mathbf{g}]\!], [\![\ell]\!]$ as output by Algorithm 5.
Output: An vector $\mathbf{a}$.
1 $\mathbf{g} := \mathsf{Dec}([\![\mathbf{g}]\!])$
2 $\ell := \mathsf{Dec}([\![\ell]\!])$
3 $\mathbf{a}[i] :=$ unset, for $i = 1, \ldots, n$
4 **while** *not all* $\mathbf{a}[i]$ *were set* **do**
`// Find a singleton`
5 find $p \in \{1, \ldots, m\}$ s.t. $\mathbf{g}[p] = 1$
`// Assign the label of the singleton`
6 $q := \ell[p]$
7 $C := \{i \mid G_{p,i} = 1\}$
8 $R := \{j \mid G_{j,q} = 1\}$
9 $\mathbf{a}[q] := 1$
10 $\mathbf{a}[i] := 0$, for $i \in C \setminus \{q\}$
11 **foreach** $i \in R$ **do**
`// Peel the contribution of` p
12 Update $\mathbf{g}[i] := \mathbf{g}[i] - 1$
13 Update $\ell[i] := \ell[i] - q$
14 **Output** $\mathbf{a}[1], \ldots, \mathbf{a}[n]$

Algorithms Overview. The algorithms get as parameters a binary generating matrix $G \in \mathbb{Z}_n^{m\times n}$ and a labeling matrix $L \in \mathbb{Z}_n^{n\times n}$ where $L_{i,i} = i$, for $i = 1, \ldots, n$ and $L_{i,j} = 0$ for $i \neq j$. The decompression algorithm is given $g = G \cdot \mathbf{a}$ and $\ell = (G \cdot L) \cdot \mathbf{a}$. Before explaining how it works, we make a few observations on g and ℓ.

Observation 2. *Using $g[p]$ we can tell whether the p-th subset is:*

- *empty - if* $\mathbf{g}[p] = 0$
- *a singleton - if* $\mathbf{g}[p] = 1$
- *none of the above - if* $\mathbf{g}[p] > 1$

Also, if the p-th subset is a singleton then $\ell[p]$ is the label of that singleton.

Algorithm 6 iterates until all elements of $\mathbf{a}$ have been recovered. In each iteration, it looks for a singleton, i.e. p such that $\mathbf{g}[p] = 1$ (Line 5). Set q to be the label of the singleton (Line 6); set C to be the set of indices in the p-th subset, which is given by the nonzero elements in the p-the column of G (Line 7); and set L to be the subset indexes that the p-element appears in, which is given by the nonzero elements in the q-th row of G (Line 8). Next, the algorithm sets the label of the singleton, $\mathbf{a}[q] = 1$ (Line 9) and resets the other elements in the subset (Line 10). Finally, the algorithm removes ("peels") the contribution of $\mathbf{a}[q]$ from every subset that q is part of. This is done by decreasing $\mathbf{g}[i]$ (Line 12) and subtracting q from $\ell[i]$ (Line 13).

Theorem 5.2. *A s-sparse vector $\mathbf{a}$ of size n can be compressed to 2 vectors $\mathbf{g}, \ell \in \mathbb{Z}_n^m$ using a circuit of size $O(n \cdot m)$ and multiplicative depth of $O(1)$, such that $\mathbf{a}$ can be reconstructed in cleartext in time $O(n + ms)$ time with probability at least $1 - s\left(1 - \frac{\binom{n-s}{n/s-1}}{\binom{n}{n/s}}\right)^m$.*

The proof is given in Appendix A.4

Moving from $\mathbb{Z}_n$ to $\mathbb{Z}_2$. Let $G \in \mathbb{Z}_2^{m\times n}$ be a binary generating matrix as before, and $H \in \mathbb{Z}_2^{k\times n}$ be a matrix with the property that every s columns are independent. Denote by G^H the $km \times n$ matrix we get by expanding each element $G_{i,j}$ to $H_i \cdot G_{i,j}$, where H_i is the i-th column of H.

$$G^L = \begin{pmatrix} (H_1) \cdot G_{1,1} & \cdots & (H_n) \cdot G_{m,n} \\ \vdots & \ddots & \vdots \\ (H_1) \cdot G_{1,n} & \cdots & (H_n) \cdot G_{m,n} \end{pmatrix}$$

The compression then calculates and outputs $[\![\mathbf{h}]\!] = G^H \cdot [\![\mathbf{a}]\!]$, and $\mathbf{h}$ is of length $m \cdot k$. We group the elements of $\mathbf{h}$ into m batches of size k that we denote as $\mathbf{h}_1, \ldots, \mathbf{h}_m$, where $\mathbf{h}_b = (\mathbf{h}[(b-1)k + 1], \ldots, \mathbf{h}[bk])$.

Observation 3. *Using $\mathbf{h}_p$ we can tell whether the p-th subset is:*

- *empty - if* $\mathbf{h}_p = (0, \ldots, 0)$
- *a singleton - if $\mathbf{h}_p$ is a column of H*
- *non of the above - otherwise*

If $\mathbf{h}_p$ is a column of H (singleton) then its index in H is the label of the singleton.

In this case, Algorithm 6 needs to change as follows:

- The search for a singleton (Line 5) becomes:
  find $p \in \{1, \ldots, m\}$ s.t. $\mathbf{h}_p$ is a column of H
- Setting the label (Line 6) becomes:
  $q :=$ the index of $\mathbf{h}_p$ in H

- Peeling the singleton (Lines 12-13) becomes:

$$\text{Update } \mathbf{h}_i := \mathbf{h}_i - \mathbf{h}_p$$

THEOREM 5.3. *A s-sparse vector $\mathbf{a}$ of size n can be compressed to an vector $\mathbf{b} \in \mathbb{Z}_2^{mk}$ using a circuit of size $O(n \cdot mk)$ and multiplicative depth of $O(1)$, such that $\mathbf{a}$ can be reconstructed in cleartext in time $O(nmk)$ with probability at least $\left(1 - 2^{s-k}\binom{n}{s+1}\right) \cdot \left(1 - s\left(1 - \frac{\binom{n-s}{n/s-1}}{\binom{n}{n/s}}\right)^m\right)$.*

The proof is given in Appendix A.5.

6 Experiments

To test our algorithms we implemented and ran them. In this section we describe the experiments we made on our main contribution (Alg. 1 and 2) and the results we got. The experiments for the other algorithms (Alg. 5 and Alg. 3) are given in Appendix B.

The system. Our algorithms are generic but we implemented them with FHE. We tested our algorithms on a system with CPUs and on a system with a GPU. We used the HElayers library [1] that wraps multiple cryptographic libraries and provides an easier unified API for them. For the experiments done on CPU we used the CKKS [10] and TFHE [12] implementations of OpenFHE [6] and ran the experiments on a server with 4 Intel(R) Xeon(R) Platinum 8260 cores running at 2.40GHz. For the experiments done on GPU we used the CKKS implementation of HElayers and ran it on NVIDIA A100 SXM4 with 80GB.

The input to our compression algorithms was s-sparse binary vectors of size n. We made 2 sets of experiments. In one set we fixed n and tried various values for s. Here we used $s = 3, \ldots, 10$. In another set we fixed s and used various values of n. Here n set to be between 16 to 512 ciphertexts.

The cryptographic schemes: Different implementations of CKKS support different sets of parameters. On CPU we used CKKS with these parameters: cyclotomic polynomial of 2^{17} degree (2^{16} slots), multiplication depth = 20, integer precision = 7, fractional precision = 48 and security level = 128. On GPU we used CKKS with these parameters: cyclotomic polynomial of 2^{16} degree (2^{15} slots), multiplication depth = 11, integer precision = 15, fractional precision = 42 and security level = 128. These configurations support bootstrapping. We used HElayers' feature to auto-bootstrap ciphertexts when they reach the lowest level.

Since *copyRecurseComp* and *copyRecurseDecomp* are deep, it is interesting to compare their implementation in CKKS and in TFHE: on one hand TFHE has a more efficient bootstrap operation but on the other hand CKKS has multiple slots in each ciphertext (2^{16} in our case). It was already shown that when batching multiple inputs CKKS can be more efficient (e.g., for AES [2]). Also, for large inputs CKKS can be more efficient (e.g., for game of life [15]).

Packing: As mentioned, CKKS operates as Single Instruction Multiple Data (SIMD), i.e. each ciphertext has multiple slots (2^{16} or 2^{15} in our case) that hold multiple values. Each operation is carried slot-wise. To allow a fair comparison between TFHE and CKKS when running *copyRecurseComp* and *copyRecurseDecomp* we packed a batch of c vectors, where c is the number of slots the key was configured to (either 2^{15} or 2^{16} depending on what the scheme implementation supported), and compressed them in parallel. We feel this is a fairer comparison because in CKKS only whole ciphertexts (with all slots) can be stored or communicated.

What we tested: We tested the time it takes to compress a s-sparse vector of size n. We also counted the size of the output, i.e. how many ciphertexts the input was compressed to. We also measured the time to run our copy-and-recurse decompression algorithm (*copyRecurseDecomp*, Algorithm 2) and compared it to the folklore implementation (described in details in Appendix C).

The results for *copyRecurseComp* (Alg. 1) and *copyRecurseDecomp* (Alg. 2) are shown in these figures:

- Figure 7 shows TFHE on CPU when s is fixed
- Figure 9 shows CKKS on CPU when s is fixed
- Figure 8 shows TFHE on CPU when n is fixed
- Figure 10 shows CKKS on CPU when n is fixed
- Figure 5 shows CKKS on GPU when n is fixed (also compares to the folklore implementation)
- Figure 6 shows CKKS on GPU when n is fixed

The tables with the raw values are given in Appendix B.

6.1 Discussion:

copyRecurseComp takes more time. Compressing a vector with *copyRecurseComp* takes more time than other algorithms. For example, running with $n = 128 \cdot 2^{15}$ and $s = 3$ takes on GPU 47 seconds with *copyRecurseComp* and only 8 seconds with the folklore algorithm. This is because *copyRecurseComp* is significantly more complicated to allow the decompression to run efficiently under FHE.

Comparing times of compression and decompression. We see that *copyRecurseDecomp* takes more time than *copyRecurseComp*, but only in CKKS and when $n > 64 \cdot 2^{16}$. This is an artifact of the bootstrapping operation. In TFHE, a bootstrap operation is performed after each multiplication and so the number of bootstraps is roughly the same in *copyRecurseComp* and in *copyRecurseDecomp*. In CKKS (with our parameters), on the other hand, ciphertexts start at chain-index $CI = 45$, i.e. supporting 45 multiplications before needing a bootstrap. When $n \leq 64 \cdot 2^{16}$ this chain-index is enough to run *copyRecurseComp* and then *copyRecurseDecomp* without needing to bootstrap. As n grows, the chain-index with which we reach *copyRecurseDecomp* is smaller and we require more bootstraps in *copyRecurseDecomp*.

Another interesting observation is that *copyRecurseDecomp* under CKKS is a little faster than *copyRecurseComp* when $n < 64 \cdot 2^{16}$. This is because when n is small enough to allow execution of *copyRecurseComp* and then *copyRecurseDecomp* without requiring bootstrapping, then operations in *copyRecurseDecomp* are executed with a smaller chain-index. This involves less ciphertext limbs and therefore are faster.

Comparing CKKS and TFHE. is a bit tricky because a ciphertext in TFHE carries a single message while a ciphertext in CKKS carries 2^{16} messages (with the parameters of our CPU implementation). For example, when $n = 256$ and $s = 3$, *copyRecurseComp* (and similarly for *copyRecurseDecomp*) under TFHE takes 1,043 seconds to compress to 41 ciphertexts. This input fits into a single CKKS ciphertext and compression is meaningless. Compressing a batch of 2^{16} vectors each with $n = 256$ and $s = 3$ takes 11,680 seconds under CKKS, which is an amortized time of 0.17 seconds per batch. We conclude that in terms of *latency* (i.e., the time to get the first output in a batch) TFHE is more favorable, but in terms of *throughput* (i.e., the time to get the output of an entire batch) CKKS is more favorable. This is consistent with comparing CKKS and TFHE on other problems (e.g., AES [2], game-of-life [15]).

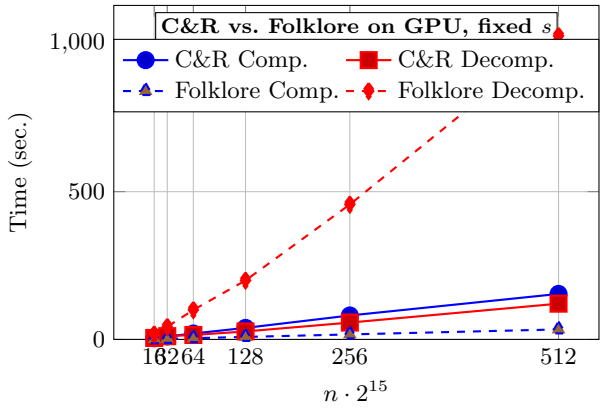

Figure 5: Comparing *copyRecurseComp* (Alg. 1) and *copyRecurseDecomp* (Alg. 2) to the baseline folklore algorithms (Alg. 7 and Alg. 8) under CKKS on GPU. Here we fixed $s = 3$ and set different sizes of the vector.

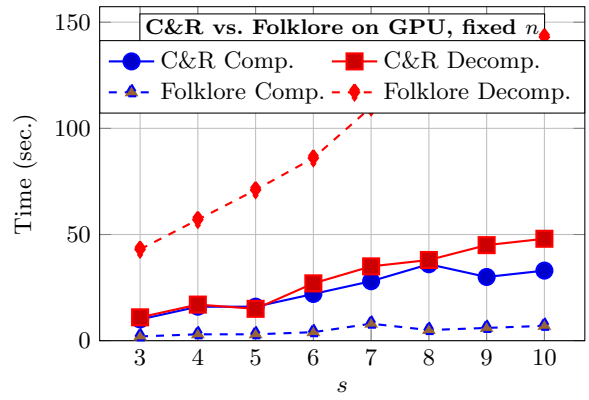

Figure 6: Comparing *copyRecurseComp* (Alg. 1) and *copyRecurseDecomp* (Alg. 2) to the baseline folklore algorithms (Alg. 7 and Alg. 8) under CKKS on GPU. Here we fixed $n = 32 \cdot 2^{15}$ and set different sparsity.

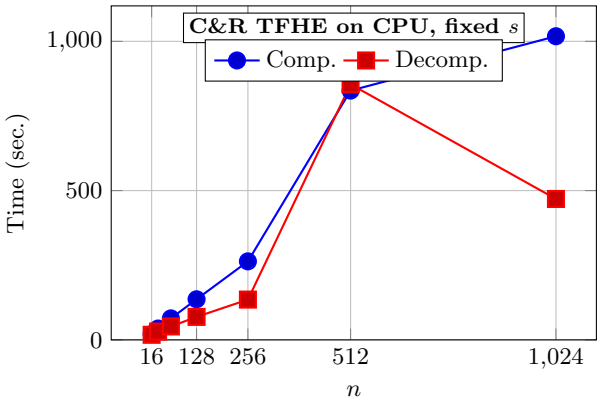

Figure 7: Running *copyRecurseComp* (Algorithm 1) and *copyRecurseDecomp* (Algorithm 2) both under TFHE, on 3-sparse vectors of different sizes, where n is the size of the vector.

7 Conclusions and Open Problems

In this paper, we showed the first (to the best of our knowledge) FHE friendly compression algorithm that has a matching FHE friendly decompression algorithm. These algorithms can be used to reduce communication or storage in systems such as agentic systems. The algorithms we show here compress s-sparse vectors. The main idea is to build a binary tree in which the elements of the vector are mapped to the leaves and then traverse s paths from the root to the nonzero leaves. This traversal is done efficiently using the copy-and-recurse technique that was recently introduced.

In addition, we observed the connection between compression and error-correcting codes. Although this observation is well known, all previous work on compression under FHE used tailored solutions and did not use the vast research already done in ECC. We find this observation interesting on its own for 2 reasons: (1) algorithms under FHE can be improved using existing ECC research; (2) the inherent constraints of FHE motivate ECC research of questions that were previously trivial. Inspired by ECC we also gave 2 novel compression algorithms. One inspired by linear-codes and the other inspired by LDPC.

Open problems: We leave these open problems for future work:

(1) Show explicit reduction from specific low-entropy inputs to s-sparse vectors.
(2) Design an error-correcting code that communicates elements in $\mathbb{Z}_n$ (with $n > 2$).
(3) The Low-rate regime was ignored in cleartext, but under FHE some interesting sparse codes were developed. Can these codes be useful in cleartext ECC? Can ideas from cleartext ECC improve codes under FHE?

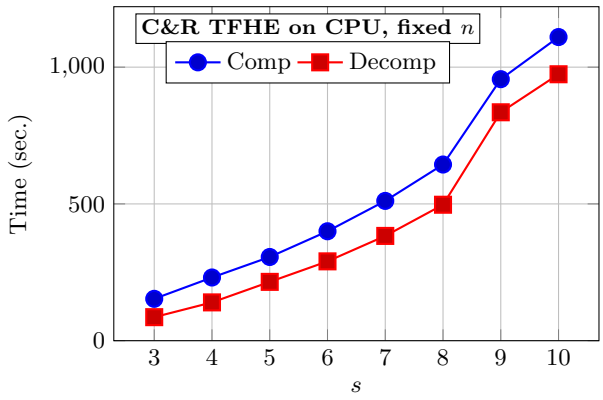

Figure 8: Running *copyRecurseComp* (Algorithm 1) and *copyRecurseDecomp* (Algorithm 2) both under TFHE, on s-sparse vectors size 128.

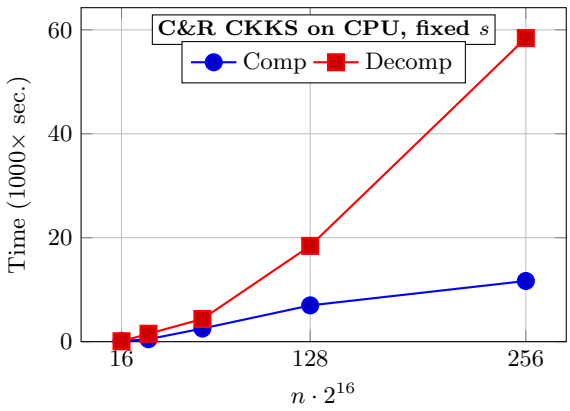

Figure 9: Running *copyRecurseComp* (Algorithm 1) and *copyRecurseDecomp* (Algorithm 2) both under CKKS, on 3-sparse vectors of different sizes, where n is the size of the vector.

(4) In Section 5.1.1 we show a simple low-rate code. Can this code be improved?
(5) The analysis in Section 5.2 for the algorithm inspired by LDPC is very simplistic. Can belief-propagation improve the decompression algorithm or the size of the output?
(6) Can more algorithms under FHE be inspired by ECC codes?
(7) Can existing work in ECC improve previous work asymptotically or by constant factors?

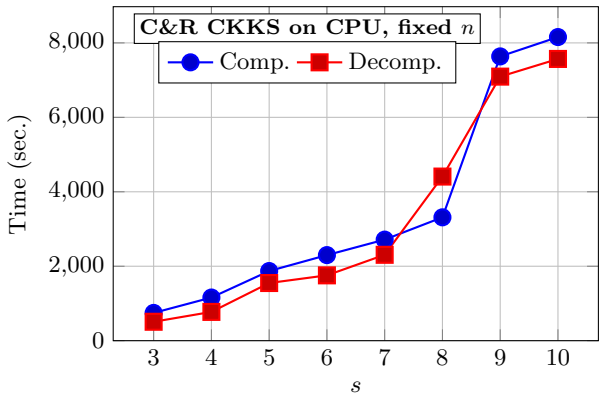

Figure 10: Running *copyRecurseComp* (Algorithm 1) and *copyRecurseDecomp* (Algorithm 2) under CKKS on s-sparse vectors of size $n = 32 \cdot 2^{16}$.

References

[1] Ehud Aharoni, Allon Adir, Moran Baruch, Nir Drucker, Gilad Ezov, Ariel Farkash, Lev Greenberg, Ramy Masalha, Guy Moshkowich, Dov Murik, Hayim Shaul, and Omri Soceanu. 2023. HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data. *Privacy Enhancing Technology Symposium (PETs) 2023* (2023). https://petsymposium.org/popets/2023/popets-2023-0020.php

[2] Ehud Aharoni, Nir Drucker, Gilad Ezov, Eyal Kushnir, Hayim Shaul, and Omri Soceanu. 2023. E2E near-standard and practical authenticated transciphering. Cryptology ePrint Archive, Paper 2023/1040. https://eprint.iacr.org/2023/1040

[3] Adi Akavia, Dan Feldman, and Hayim Shaul. 2018. Secure Search on Encrypted Data via Multi-Ring Sketch. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (Toronto, Canada) *(CCS '18)*. Association for Computing Machinery, New York, NY, USA, 985–1001. doi:10.1145/3243734.3243810

[4] Adi Akavia, Dan Feldman, and Hayim Shaul. 2019. Secure Data Retrieval on the Cloud: Homomorphic Encryption meets Coresets. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2019, 2 (Feb. 2019), 80–106. doi:10.13154/tches.v2019.i2.80-106

[5] Erdal Arikan. 2009. Channel Polarization: A Method for Constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels. *IEEE Transactions on Information Theory* 55, 7 (2009), 3051–3073. doi:10.1109/TIT.2009.2021379

[6] Ahmad Al Badawi, Andreea Alexandru, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Carlo Pascoe, Yuriy Polyakov, Ian Quah, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. 2022. OpenFHE: Open-Source Fully Homomorphic Encryption Library. Cryptology ePrint Archive, Paper 2022/915. https://eprint.iacr.org/2022/915

[7] C. Berrou, A. Glavieux, and P. Thitimajshima. 1993. Near Shannon limit error-correcting coding and decoding: Turbo-codes. 1. In *Proceedings of ICC '93 - IEEE International Conference on Communications*, Vol. 2. 1064–1070 vol.2. doi:10.1109/ICC.1993.397441

[8] Alexander Bienstock, Sarvar Patel, Joon Young Seo, and Kevin Yeo. 2024. Batch PIR and labeled PSI with oblivious ciphertext compression. In *Proceedings of the 33rd USENIX Conference on Security Symposium* (Philadelphia, PA, USA) *(SEC '24)*. USENIX Association, USA, Article 333, 18 pages.

[9] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. 2012. (Leveled) Fully Homomorphic Encryption without Bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference* (Cambridge, Massachusetts) *(ITCS '12)*. Association for Computing Machinery, New York, NY, USA, 309–325. doi:10.1145/2090236.2090262

[10] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *Advances in Cryptology – ASIACRYPT 2017*, Tsuyoshi Takagi and Thomas Peyrin (Eds.). Springer International Publishing, Cham, 409–437.

[11] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. 2020. Efficient Homomorphic Comparison Methods with Optimal Complexity. In *Advances in Cryptology – ASIACRYPT 2020*, Shiho Moriai and Huaxiong Wang (Eds.). Springer International Publishing, Cham, 221–256.

[12] Ilaria Chillotti, Nicolas Gama, Mariya Georgieva, and Malika Izabachène. 2017. TFHE: Fast Fully Homomorphic Encryption over the Torus. In *Advances in Cryptology – ASIACRYPT 2017 (Lecture Notes in Computer Science, Vol. 10624)*. Springer, 307–331. doi:10.1007/978-3-319-70697-9_11

[13] Seung Geol Choi, Dana Dachman-Soled, S. Dov Gordon, Linsheng Liu, and Arkady Yerukhimovich. 2021. Compressed Oblivious Encoding for Homomorphically Encrypted Search. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS 2021)*, Giovanni Vigna and Elaine Shi (Eds.). ACM Press, 2277–2291.

[14] Yevgeniy Dodis and Thomas Shrimpton (Eds.). 2022. *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part I*. Lecture Notes in Computer Science, Vol. 13507. Springer. doi:10.1007/978-3-031-15802-5

[15] Nir Drucker, Guy Moshkowich, Tomer Pelleg, and Hayim Shaul. 2023. BLEACH: Cleaning Errors in Discrete Computations Over CKKS. *Journal of Cryptology* 37, 1 (2023), 3. doi:10.1007/s00145-023-09483-1

[16] Nils Fleischhacker, Kasper Green Larsen, and Mark Simkin. 2023. How to Compress Encrypted Data. In *Advances in Cryptology – EUROCRYPT 2023: 42nd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Lyon, France, April 23-27, 2023, Proceedings, Part I* (Lyon, France). Springer-Verlag, Berlin, Heidelberg, 551–577. doi:10.1007/978-3-031-30545-0_19

[17] R. Gallager. 1962. Low-density parity-check codes. *IRE Transactions on Information Theory* 8, 1 (1962), 21–28. doi:10.1109/TIT.1962.1057683

[18] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2021. Oblivious Key-Value Stores and Amplification for Private Set Intersection. In *Advances in Cryptology – CRYPTO 2021 - 41st Annual International Cryptology Conference, CRYPTO 2021, Proceedings (Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics))*, Tal Malkin and Chris Peikert (Eds.). Springer Science and Business Media Deutschland GmbH, Germany, 395–425. doi:10.1007/978-3-030-84245-1_14

[19] Michael T. Goodrich and Michael Mitzenmacher. 2011. Invertible Bloom Lookup Tables. In *Proceedings of the 49th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*. IEEE Computer Society Press, 792–799. doi:10.1109/Allerton.2011.6120248

[20] Ankit Gupta and Sergio Verdu. 2009. Operational duality between lossy compression and channel coding: Channel decoders as lossy compressors. In *2009 Information Theory and Applications Workshop*. 119–123. doi:10.1109/ITA.2009.5044932

[21] David A. Huffman. 1952. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the IRE* 40, 9 (1952), 1098–1101. doi:10.1109/JRPROC.1952.273898

[22] Eyal Kushnir, Guy Moshkowich, and Hayim Shaul. 2024. Secure Range-Searching Using Copy-And-Recurse. *Proceedings on Privacy Enhancing Technologies* 2024, 3 (2024), 626–644. doi:10.56553/popets-2024-0096

[23] Eyal Kushnir and Hayim Shaul. 2026. Improved Range Searching and Range Emptiness Under FHE Using Copy-and-Recurse. In *Cyber Security, Cryptology, and Machine Learning*, Adi Akavia, Shlomi Dolev, Anna Lysyanskaya, and Rami Puzis (Eds.). Springer Nature Switzerland, Cham, 35–52.

[24] C. E. Shannon. 1948. A Mathematical Theory of Communication. *The Bell System Technical Journal* 27 (1948), 379–423.

[25] A. Shokrollahi. 2006. Raptor codes. *IEEE Transactions on Information Theory* 52, 6 (2006), 2551–2567. doi:10.1109/TIT.2006.874390

[26] Madhu Sudan and Tali Kaufman. 2007. Sparse Random Linear Codes are Locally Decodable and Testable . In *2007 48th Annual IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society, Los Alamitos, CA, USA, 590–600. doi:10.1109/FOCS.2007.8

[27] Ian H. Witten, Radford M. Neal, and John G. Cleary. 1987. Arithmetic coding for data compression. *Commun. ACM* 30, 6 (June 1987), 520–540. doi:10.1145/214762.214771

[28] J. Ziv and A. Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977), 337–343. doi:10.1109/TIT.1977.1055714

A Proofs

A.1 Proof Of Theorem 4.1

Proof. Algorithm 1 starts by building the tree T (Lines 2-5). This is done from bottom to top and can be done using a circuit of size $O(n)$ and depth $O(\log \frac{n}{s})$. Then the algorithm traverses s paths $\pi = \{\pi_1, \ldots, \pi_s\}$ in T from root to bottom as we now elaborate.

At level ℓ, the algorithm keeps s subtrees whose leaves contain all nonzero leaves. It starts at level $\log s$ (all levels above it have less than s nodes and can be ignored) and moves downwards by picking the s subtrees that should be kept at level $\ell + 1$. The algorithm therefore performs $\log n - \log s$ iterations. In the iteration at level ℓ, we consider the $2s$ children of the roots (Line 9), which is denoted by $X^{(\ell)}$ (out of which at most s are set) and use them to build a selection matrix $S^{(\ell)}_{s\times 2s}$ (Line 11). This can be done with a subcircuit of size $O(s^3)$ and depth $O(s \log s)$. The algorithm selects the s children to follow by multipling $S^{(\ell)}$ by $X^{(\ell)}$. We remind that each element in $X^{(\ell)}$ is a subtree of height ℓ, and this multiplication can be done by a circuit of size $O(s^2 2^\ell)$ and depth $O(1)$.

Summing, we get that all iterations can be computed by a circuit of size $\sum_{\log s}^{\log n} \left(O\left(s^3\right) + O\left(s^2 2^\ell\right)\right) = O(s^3 \log \frac{n}{s} + \frac{n}{s} s^2) = O(s^3 \log \frac{n}{s} + ns)$, which is also the circuit size that computes Algorithm 1. The depth of the circuit is $O(s \log \frac{n}{s} \log s)$ □

A.2 Proof Of Theorem 4.2

Proof. The proof is similar to that of Theorem 4.1. Algorithm 2 builds the tree T from the leaves up. It does that by following s paths $\pi = \{\pi_1, \ldots, \pi_s\}$ as we now elaborate.

The algorithm starts at the leaves, and at level ℓ it keeps a copy of s subtrees that are rooted at nodes in level ℓ that appear on the paths in π (padding with empty subtrees if needed). The algorithm stops when it reaches level $\log s$ and all leaves have been constructed.

In the iteration at level ℓ we build an expansion matrix $Z^{(\ell)}_{2s\times s}$ (Line 5) that is the inverse of $S^{(\ell)}_{s\times 2s}$ computed by Algorithm 1. This can be done with a subcircuit of size $O(s^3)$ and depth $O(s \log s)$. The algorithm expands the s subtrees by multiplying $Z^{(\ell)}$ by $T^{(\ell)}$ (Line 6). We remind that each element in $T^{(\ell)}$ is a subtree of height ℓ, and this multiplication can be done by a circuit of size $O(s^2 2^\ell)$ and depth $O(1)$. The algorithm then unite sibling subtrees to construct the subtrees to be used in the next iteration (Line 8-9).

Summing, we get that all iterations we can be computed by a circuit of size $\sum_{\log s}^{\log n} \left(O\left(s^3\right) + O\left(s^2 2^\ell\right)\right) = O(s^3 \log \frac{n}{s} + ns)$, which is also the circuit size that computes Algorithm 2. The depth of the circuit is $O(s \log \frac{n}{s} \log s)$. □

A.3 Proof Of Theorem 5.1

PROOF. **Correctness.** In Line 8 Algorithm 4 restricts M to a regular $|C| \times |C|$ matrix. Here there is an underlying assumption that such $M''^{(j)}$ exists. We first prove that Algorithm 4 is correct assuming that a regular matrix $M''^{(j)}$ exists for all j. Then we prove that the probability of that event is $|\mathbb{F}|^{(2s-m-1)\log\frac{n}{m}}$.

We start by proving that each $\mathbf{x}_i^{(j)}$ is computed correctly, for $\log m \leq j \leq \log n$ and $1 \leq i \leq 2^j$. Recall that

$$\mathbf{x}_i^{(j)} = \begin{cases} 1, & \text{The } i\text{th node in } L_j \text{ is nonzero} \\ 0, & \text{Otherwise} \end{cases}.$$

The correctness of $\mathbf{x}^{(\log m)} = (\mathbf{x}_1^{(\log m)}, \ldots, \mathbf{x}_m^{(\log m)})$ is trivial since they are given as input. Assuming the correctness of $\mathbf{x}^{(j-1)}$ (and the existence of a regular matrix $M''^{(j)}$) we prove that $\mathbf{x}^{(j)}$ is computed correctly. From Observation 1 we get that $|C| \leq 2s$. Moreover, the children of nodes with zero value are also white and there are at least $2^j - |C|$ such nodes in $\mathbf{x}^{(j)}$. Since $\mathbf{y^{(j)}} = M^{(j)} \cdot \mathbf{x}^{(j)}$, we can remove the columns of $M^{(j)}$ that are known to be multiplied by zero elements of $\mathbf{x}^{(j)}$ and get $\mathbf{y^{(j)}} = M'^{(j)} \cdot \mathbf{x}'^{(j)}$, where $M'^{(j)} \in \mathbb{F}^{m \times |C|}$ and $\mathbf{x}'^{(j)} \in \mathbb{F}^{|C|}$. By our assumption, we can remove $m - |C|$ rows of $M'^{(j)}$ to get a regular matrix $M''^{(j)} \in \mathbb{F}^{|C| \times |C|}$. Now $\mathbf{y}'^{(j)} = M''^{(j)} \cdot \mathbf{x}'^{(j)}$, where $\mathbf{y}'^{(j)} \in \mathbb{F}^{|C|}$ is received by removing $m - |C|$ elements of $\mathbf{y}'^{(j)}$. Since $M''^{(j)}$ is regular we get $\mathbf{x}'^{(j)} = (M''^{(j)})^{-1} \cdot \mathbf{y}'^{(j)}$, which gives us the values of the nonzero nodes in L_j. This can be easily expanded to $\mathbf{x}^{(j)}$ since the nodes not in $\mathbf{x}'^{(j)}$ are known to be zero. Since $\mathbf{x}^{(\log n)} = \mathbf{a}$ we conclude that Algorithm 4 correctly compute $\mathbf{a}$ (assuming $M''^{(j)}$ exist for all $\log m + 1 \leq j \leq \log n$, something that happens with high probability as we prove next).

Before continuing to prove the success probability we reiterate (see for example [18].) an observation on the probability that a $r \times c$ matrix over a finite field, with $c \geq r$ has rank r.

OBSERVATION 4. *Let M a random matrix over $\mathbb{F}$ with r rows and $c \geq r$ columns, then M has a probability of at least $1 - |\mathbb{F}|^{r-c-1}$ to have rank r.*

In fact, as was also mentioned by [18], a similar bound holds if $r \geq c$ because $rank(M) = rank(M^T)$.

PROOF. The proof (also appearing in [18]) sums the probability of the non-overlapping events:

$$\sum_{j=1}^{r} \Pr[\text{row } j \in \text{ span of first } (j-1) \text{ rows} \mid \text{ first } j-1 \text{ rows lin. ind.}] = \sum_{i=0}^{r-1} \frac{|\mathbb{F}|^i}{|\mathbb{F}|^c} = \frac{1}{|\mathbb{F}|^c} \cdot \frac{|\mathbb{F}|^r - 1}{|\mathbb{F}| - 1} < |\mathbb{F}|^{r-c-1}.$$

□

Success probability. We now turn to compute the probability in which Algorithm 4 succeeds. This happens when for each $\log m + 1 \leq j \leq \log n$ we have $rank(M'^{(j)}) = |C_j|$ (i.e. there exists a regular $M''^{(j)} \in \mathbb{F}^{|C_j| \times |C_j|}$, that is a submatrix of $M'^{(j)} \in \mathbb{F}^{m \times |C_j|}$). From Observation 4 it follows that a random matrix over $\mathbb{F}$ with $|C_j|$ rows and $m \geq |C_j|$ columns has a probability of $|\mathbb{F}|^{|C_j|-m-1}$ to be regular. The probability that a regular sub-matrix exists for each $\log m + 1 \leq j \leq \log n$ is therefore: $\Pr[\text{all } M''^{(j)} \text{ exist}] = \prod_{\log m+1}^{\log n} |\mathbb{F}|^{C_j - m - 1} \leq |\mathbb{F}|^{(2s-m-1)\log\frac{n}{m}}$. The inequality holds since $C_j \leq 2s$.

Plaintext-multiplications. In level L_i (for $i = \log n, \ldots, \log m + 1$) the algorithm multiplies a ciphertext vector (of size 2^i) by a plaintext $m \times 2^i$ matrix (Line 10). In level L_i that can be done using $m2^i$ plaintext-multiplications for a total of $2m(n-m)$ plaintext-multiplications.

Communication Size. The output of Algorithm 3 is

$$S^{(\log m+1)}, \ldots, S^{(\log n)}, [\![\mathbf{y}^{(\log m)}]\!], \ldots, [\![\mathbf{y}^{(\log n)}]\!]$$

. Remembering that $y^{(i)}$ is a vector of m elements, this adds up to $m \log \frac{n}{m}$ ciphertexts and $\log \frac{n}{m}$ seeds.

□

A.4 Proof Of Theorem 5.2

PROOF. Construct G by drawing each column uniformly from the set of all n/s-sparse n-dim vectors.

Compression circuit size and depth. Compression involves a multiplication of $m \times n$ matrix by a vector. This can be done with a circuit of size $O(mn)$ and depth $O(1)$.

Decompression time. The decompression makes s iterations where in each iteration it: (1) Looks for p such that $\mathbf{g}[p] = 1$ (done in $O(m)$ time); (2) Peels the contribution of p (done in $O(n/s)$ time). The total time for s iterations is therefore, $O(n + ms)$

Success probability. It is easy to see that each column $G[i]$ is a singleton with probability $p = \frac{\binom{s}{1}\cdot\binom{n-s}{n/s-1}}{\binom{n}{n/s}}$. Set X_m to be the random variable whose value is the number of distinct singletons in G. Using the principle of inclusion-exclusion we get

$$Pr[X_k = s \mid \text{ all } k \text{ columns are singletons}] = \sum_{j=0}^{s} (-1)^j \binom{s}{j} \left(1 - \frac{j}{s}\right)^k.$$

Extending this to m columns we have

$$Pr[X_m = s] = \sum_{k=s}^{m} \binom{m}{k} p^k (1-p)^{m-k} \sum_{j=0}^{s} (-1)^j \binom{s}{j} \left(1 - \frac{j}{s}\right)^k \geq \sum_{k=s}^{m} \binom{m}{k} p^k (1-p)^{m-k} \left(1 - s\left(1 - \frac{1}{s}\right)^k\right) = 1 - s\sum_{k=s}^{m} \binom{m}{k} \left(p - \frac{p}{s}\right)^k (1-p)^{m-k} \geq 1 - s\sum_{k=0}^{m} \binom{m}{k} \left(p - \frac{p}{s}\right)^k (1-p)^{m-k} = 1 - s(1 - \frac{p}{s})^m. \quad (1)$$

Plugging the probability p we get,

$$Pr[X_m = s] \geq 1 - s\left(1 - \frac{\binom{s}{1}\binom{n-s}{n/s-1}}{s\binom{n}{n/s}}\right)^m = 1 - s\left(1 - \frac{\binom{n-s}{n/s-1}}{\binom{n}{n/s}}\right)^m \quad (2)$$

□

A.5 Proof Of Theorem 5.3

Proof. Construct G as in the proof of Theorem 5.2, and construct H by drawing each element uniformly from $\mathbb{Z}_2$.

Compression circuit size and depth. The compression algorithm involves a multiplication of a matrix by a vector which can be done in a circuit of size $O(mkn)$ and depth of $O(1)$.

Decompression time. The decompression algorithm performs s iterations, where in each iteration it: (1) Looks for p such that b_p is a singleton. This can be done in $O(mk \log n)$ if the columns of H are sorted. (2) Peels the contribution of p in $O(mkn/s)$ time. For a total of $O(mkn)$ for s iterations.

Success probability. G is created in the same way as in the proof of Theorem 5.2 with the same success probability. It remains to bound the probability that a randomly drawn H has the property that any $s+1$ columns are linearly independent, which we show next.

Using Observation 4, the probability that a specific subset of $s+1$ columns of H is independent is: $Pr[s+1 \text{ columns are independent}] > 1-2^{s-k}$ and the probability that all $\binom{n}{s+1}$ choices of columns are independent is $Pr[\text{every set of } s+1 \text{ columns are independent}] > 1-\binom{n}{s+1}(2^{s-k})$. The success probability of Algorithm 6 is therefore:

$$Pr[\text{Algorithm succeeds}] > \left(1-2^{s-k}\binom{n}{s+1}\right)\cdot\left(1-s\left(1-\frac{\binom{n-s}{n/s-1}}{\binom{n}{n/s}}\right)^{m}\right).$$

□

Scheme	n	s	Comp.	Decomp.	Comm.
CKKS	$16 \cdot 2^{16}$	3	108	84	$17 \cdot 2^{16}$
	$32 \cdot 2^{16}$		478	1,525	$23 \cdot 2^{16}$
	$64 \cdot 2^{16}$		2,528	4,369	$29 \cdot 2^{16}$
	$128 \cdot 2^{16}$		6,981	18,425	$35 \cdot 2^{16}$
	$256 \cdot 2^{16}$		11,680	58,423	$41 \cdot 2^{16}$
	$32 \cdot 2^{16}$	3	746	508	$23 \cdot 2^{16}$
		4	1,159	773	$28 \cdot 2^{16}$
		5	1,871	1,549	$31 \cdot 2^{16}$
		6	2,299	1,755	$34 \cdot 2^{16}$
		7	2,715	2,307	$37 \cdot 2^{16}$
		8	3,314	4,409	$40 \cdot 2^{16}$
		9	7,637	7,094	$41 \cdot 2^{16}$
		10	8,157	7,569	$42 \cdot 2^{16}$
TFHE	16	3	66	66	17
	32		132	130	23
	64		270	258	29
	128		525	492	35
	256		1,043	950	41
	512		2,053	1,884	47
	1,024		4,047	3,669	53
	128	3	529	495	35
		4	831	796	44
		5	968	956	51
		6	1,292	1,284	58
		7	1,687	1,639	65
		8	2,138	2,091	72
		9	2,577	2,540	77
		10	3,021	2,939	82

Table 2: Running *copyRecurseComp* and *copyRecurseDecomp* under CKKS and TFHE schemes. The 1st block shows results for CKKS when $s=3$ is fixed. The 2nd block shows results for CKKS when $n = 32 \cdot 2^{16}$ is fixed. The 3rd block shows results for TFHE when $s=3$ is fixed. The 4th block shows results for TFHE when $n=128$ is fixed. The scheme, n and s are given in the 3 first columns. Columns 4 and 5 show the time in seconds to run *copyRecurseComp* and *copyRecurseDecomp*. The last column shows the output size.

B Full Experiment Data

In this section we give the full experiments details.

- Table 2 shows the raw full details of running copy-and-recurse on a CPU system using CKKS and TFHE.
- Table 3 shows the details of comparing copy-and-recurse with the folklore algorithms on a GPU.
- Table 4 shows the details of *treeComp* algorithm.
- Table 5 shows the details of *peelComp* algorithm.
- Figure 12 shows how the running time of *treeComp* varies with n when s is fixed.
- Figure 11 shows how the running time of *treeComp* varies with s when n is fixed.
- Figure 14 shows how the running time of *peelComp* varies with n when s is fixed.
- Figure 13 shows how the running time of *peelComp* varies with s when n is fixed.

C The Folklore Implementation

In this section we describe the folklore implementations for compressing and decompressing a s-sparse binary vector.

Folklore Compression Overview. The folklore compression algorithm (Alg. 7 gets as input an encrypted s-sparse binary vector and outputs an encrypted vector of the labels of the nonzero elements. To do that the algorithm keeps a mask vector of s elements that indicates the slot in the output that the next label should be written to. The mask is initialized with $(1, 0, \ldots, 0)$, which is the vector that indicates the first slot (Line 1). The algorithm then goes over the n elements in the input and if the input is nonzero it updates the label in the output and shifts the mask right, as we now elaborate. To update the label in the output, the algorithm creates a vector that keeps the label in the appropriate slot (Line 4). The label is added to the output only if the input element is nonzero. This is done using a MUX statement (Line 6). Eventually, the mask is shifted right if the input element is nonzero with another MUX statement (Line 10).

Folklore Decompression Overview. The folklore decompression (Alg. 8) gets as input an encrypted vector of labels (the output of Algorithm 7). It construct a binary vector whose

	$n \cdot 2^{15}$	Comp.	Decomp.	Comm.
folklore	16	1	18	3
	32	2	43	3
	64	4	100	3
	128	8	200	3
	256	17	457	3
	512	34	1028	3
Copy-and-recurse	16	4	6	17
	32	12	13	23
	64	25	35	29
	128	47	47	35
	256	118	150	41
	512	176	195	47

Table 3: Running Copy-and-Recurse and the folklore algorithms on a GPU. The first column shows the algorithm we used: the folklore (Alg. 7 and Alg. 8) or Copy-and-Recurse (Alg.1 and Alg. 2). The next 2 columns show the size (n) of the vector to compress and its sparsity (s). The next 2 columns show the times to compress and decompress in seconds. The last column shows the size of the compressed vector in ciphertexts.

n $[\cdot 2^{16}]$	s	Time	Comm. $[\cdot 2^{16}]$
1,024	3	213	29
	4	254	36
	5	316	44
	6	378	52
	7	439	60
	8	444	64
	9	499	71
	10	545	78
	11	595	85
128	3	29	20
256		42	23
512		95	26
1,024		213	29
2,048		474	32
4,096		1,045	35

Table 4: The results for running *treeComp* (Algorithm 3) with $|\mathbb{F}| \geq n$ on a s-sparse vector of size n. The table shows computation time (in seconds) and communication cost.

jth element is 1 iff its label, j, appears in the input label vector. The algorithm is quite straightforward. It computes for every pair i, j, where $i = 1, \ldots, s$ and $j = 1, \ldots, n$ whether the ith label in the input is j. It does that by using the well studied primitive

$$Sign(x) = \begin{cases} -1, & \text{if } x \leq -\varepsilon \\ 1, & \text{if } x \geq \varepsilon \end{cases}$$

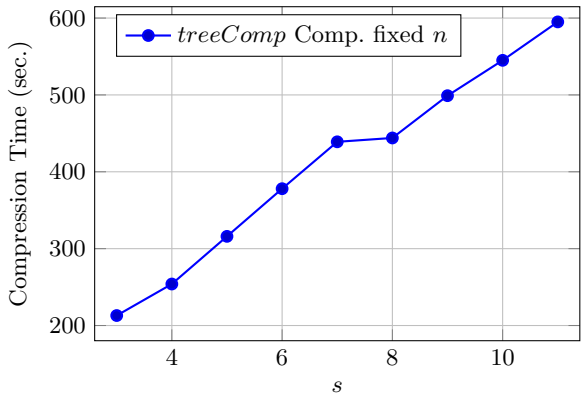

Figure 11: Time to run *treeComp* (Algorithm 3) over CKKS as a function of s with fixed n.

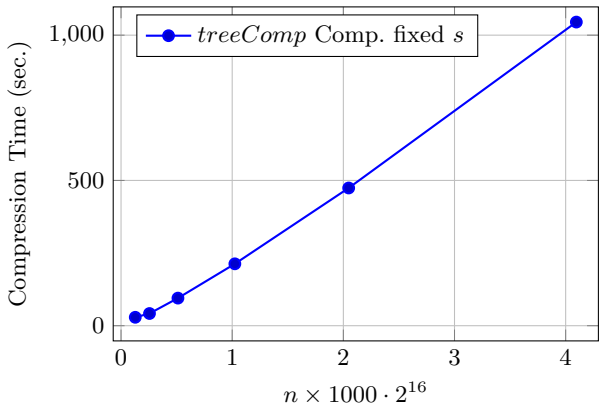

Figure 12: Time to run *treeComp* (Algorithm 3) over CKKS as a function of n with fixed s.

Algorithm 7: $naiveComp_{n,s}([\![\mathbf{a}]\!])$

mask $:= ([\![1]\!], [\![0]\!], \ldots, [\![0]\!])$ `// a` s`-dim vector`
$[\![\mathbf{o}]\!] := ([\![0]\!], \ldots, [\![0]\!])$ `// a` s`-dim vector`
foreach $i = 1, \ldots, n$ **do**
  $[\![\ell]\!] := i \cdot [\![\mathbf{mask}]\!]$
  foreach $j = 1, \ldots, s$ **do**
    $[\![\mathbf{o}[j]]\!] := [\![\mathbf{o}[j]]\!] + [\![\mathbf{a}[i]]\!] \cdot [\![\mathbf{mask}[j]]\!]$
  $[\![\mathbf{newmask}]\!] := ([\![1]\!], [\![0]\!], \ldots, [\![0]\!])$ `// a` s`-dim vector`
  $[\![\mathbf{newmask}[0]]\!] := [\![\mathbf{mask}[0]]\!] \cdot (1 - [\![\mathbf{a}[i]]\!])$
  foreach $j = 2, \ldots, s$ **do**
    $[\![\mathbf{newmask}[j]]\!] := [\![\mathbf{mask}[j]]\!] \cdot (1 - [\![\mathbf{a}[i]]\!]) + [\![\mathbf{mask}[j-1]]\!] \cdot [\![\mathbf{a}[i]]\!]$
  $[\![\mathbf{mask}]\!] := [\![\mathbf{newmask}]\!]$
Output $[\![\mathbf{o}]\!]$

defined for $-1 \leq x \leq 1$. See for example [11]. When $-\varepsilon < x < eps$ the value of $Sign(x)$ is not guaranteed and the

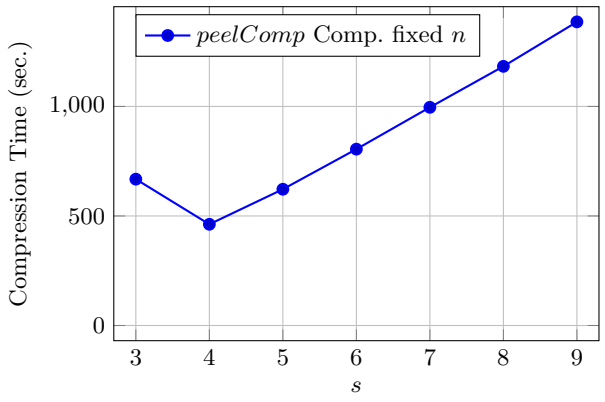

Figure 13: Running time of *peelComp* (Algorithm 5) under CKKS with a fixed $n = 1000$ for various values of s. The value of m was set to allow 90% success probability using the bound from Theorem 5.2.

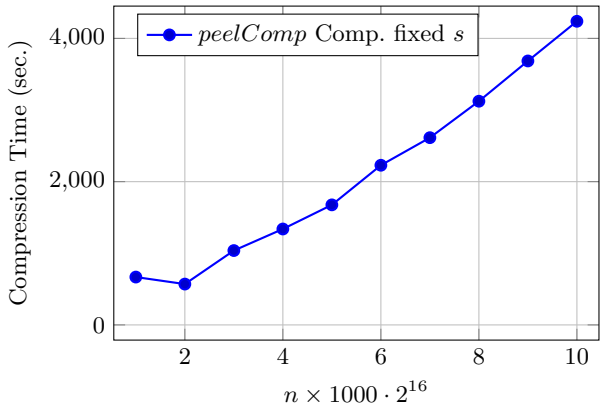

Figure 14: Time to run *peelComp* (Algorithm 5) over CKKS as a function of n with fixed s.

Algorithm 8: $naiveDecomp_{n,s}(\llbracket l \rrbracket)$

Parameters: integers n, s.
Input: A vector of s labels $\llbracket \mathbf{l} \rrbracket = (\llbracket \mathbf{l}[1] \rrbracket, \ldots, \llbracket \mathbf{l}[s] \rrbracket)$.
Output: A s-sparse binary vector with nonzero elements in the slots indicated by l.

1 $\llbracket \mathbf{o} \rrbracket := (\llbracket 0 \rrbracket, \ldots, \llbracket 0 \rrbracket)$ `// a` n`-dim vector`
2 **foreach** $i = 1, \ldots, s$ **do**
3 **foreach** $j = 1, \ldots, n$ **do**
4 $\llbracket e_{i,j} \rrbracket := 0.5 \cdot Sign(\frac{(\llbracket \mathbf{l}[i] \rrbracket - j)^2 - 0.5}{n^2}) - 0.5$
5 $\llbracket \mathbf{o}[j] \rrbracket := \llbracket \mathbf{o}[j] \rrbracket + \llbracket e_{i,j} \rrbracket$
6 **Output** $\llbracket \mathbf{o} \rrbracket$

running time grows as ε is taken smaller. in Line 4 the algorithm computes the sign function of $d = \frac{(\llbracket \mathbf{l}[i] \rrbracket - j)^2 - 0.5}{n^2}$ which is $-\frac{1}{2n^2}$ if the ith label is j and otherwise has a value $\frac{1}{2n^2} \leq d \leq 1 - \frac{1}{2n^2}$. The value of $Sign(e)$ is then normalized to 1 if $l[i] = j$ and 0 otherwise and added to the output (Line 5).

$n[\cdot 2^{16}]$	s	m	Time	Comm. $[\cdot 2^{16}]$
1,000	3	22	668	44
	4	34	462	68
	5	46	622	92
	6	59	805	118
	7	73	996	146
	8	87	1,183	174
	9	102	1,386	204
1,000	3	22	668	44
2,000		22	570	44
3,000		22	1,038	44
4,000		22	1,339	44
5,000		22	1,676	44
6,000		22	2,229	44
7,000		22	2,614	44
8,000		22	3,123	44
9,000		22	3,685	44
10,000		22	4,239	44

Table 5: The results for running *peelComp* (Algorithm 5) with $|\mathbb{F}| \geq n$ on a s-sparse vector of size n. The parameter m was chosen using the bound in Theorem 5.2 to guarantee 90% success probability when decompressing.

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009