# Long Paper: Orion's Ascent: Accelerating Hash-Based Zero Knowledge Proof on Hardware Platforms

Florian Hirner<sup>1</sup>, Florian Krieger<sup>1</sup>, Constantin Piber<sup>2</sup> and Sujoy Sinha Roy<sup>1</sup>

<sup>1</sup> IAIK, Graz University of Technology, Graz, Austria firstname.lastname@iaik.tugraz.at
<sup>2</sup> Graz University of Technology, Graz, Austria firstname.lastname@student.tugraz.at

**Abstract.** Zero-knowledge proofs (ZKPs) are cryptographic protocols that enable one party to prove the validity of a statement without revealing the underlying data. Such proofs have applications in privacy-preserving technologies and verifiable computations. However, slow proof generation poses a significant challenge in the wide-scale adoption of ZKP. Orion is a recent ZKP scheme with linear prover time. It leverages coding theory, expander graphs, and Merkle hash trees to improve computational efficiency. However, the polynomial commitment phase in Orion is yet a primary performance bottleneck due to the memory-intensive nature of expander graph-based encoding and the data-heavy hashing required for Merkle Tree generation. This work introduces several algorithmic and hardware-level optimizations aimed at accelerating Orion's commitment phase. We replace the recursive encoding construction with an iterative approach and propose novel expander graph strategies optimized for hardware to enable more parallelism and reduce off-chip memory access. Additionally, we implement an on-the-fly expander graph generation technique, reducing memory usage by gigabytes. Further optimizations in Merkle Tree generation reduce the cost of SHA3 hashing, resulting in significant speedups of the polynomial commitment phase. Our FPGA implementation heavily optimizes access to the off-chip high-bandwidth memory (HBM) utilizing memory-efficient computational strategies. The accelerator demonstrates speedups of up to  $381 \times$  for linear encoding and up to  $2,390 \times$  for the hashing operations over a software implementation on a high-end CPU. In the context of real-world applications, such as zero-knowledge proof-of-training of deep neural networks (DNNs), our techniques show up to  $241 \times$ speed up for the polynomial commitment.

**Keywords:** Zero-Knowledge Proof  $\cdot$  Orion  $\cdot$  Accelerate Commitment  $\cdot$  Scalable Architecture  $\cdot$  Hardware Accelerator

## 1 Introduction

Zero-knowledge proofs (ZKP) enable one party, known as the *prover*, to demonstrate to another party, known as the *verifier*, that a given statement is true without revealing any information about why the statement is true or any private data that was used to prove it. As an illustration, the prover can convince the verifier that it knows a private witness w for a public input x such that C(x, w) = 0 is satisfied for a circuit C, all without revealing any information about w. The concept of ZKP systems was first introduced by Goldwasser, Micali, and Rackoff in the 1980s in their seminal paper [GMR85].

ZKPs have seen growing use in recent years, with their real-life applications expected to expand further. One notable application area is verifiable computation, where a client delegates computation to a powerful but untrusted server. Subsequently, the client can easily verify (without re-doing the computation) whether the server executed the computation correctly or not. Concrete realizations of verifiable machine learning using ZKP are presented in [LXZ21, WYX<sup>+</sup>21]. ZKPs are extensively used in blockchains and cryptocurrencies to achieve anonymity and privacy [BSCG<sup>+</sup>14]. Cryptocurrency transactions can be fully encrypted on the blockchain, yet their legitimacy can still be verified using ZKP [Fou, Pro]. Other application areas include online auction [GY18], verifiable database query [LWX<sup>+</sup>23], and classical authentication systems. Recently proposed zero-knowledge proof of training framework (zkPoT) [APPK24] enables a prover to prove the correct training of a DNN on a committed dataset without disclosing any details about the model or dataset. The prover trains the model iteratively, committing to the model parameters and providing a zkPoT at each step. The framework uses the Orion [XZS22] ZKP scheme, known for its fast linear-time commitment and proof. Besides, Orion's cryptographic security is based on the preimage resistance of a cryptographic hash function. Orion's fast prover and post-quantum security sets it apart from commonly used proof systems [KZG10, Gro16, WTS<sup>+</sup>18, BBB<sup>+</sup>18, Lee21] which are based on pairings or discrete logarithm assumptions.

Motivation for hardware acceleration of Orion [XZS22]: Several works in the literature have accelerated pairing and discrete logarithm-based proof systems. However, hardware acceleration Orion (which was proposed in 2022) using FPGAs has not gained attention. Although Orion has a linear time complexity, its software implementation [XZS22] is slow. Furthermore, when Orion is used as an inner proof system in the recent zkPoT [APPK24] framework of DNN, committing to the dynamic polynomials (dependent on training data and parameters) is the most time-consuming component. A detailed timing analysis in Subsection 3.3 of the software shows that the most computationally intensive part of Orion is the polynomial commitment phase, where large expander graphs are used in the encoding of a linear error-correcting code. This phase involves recursive encoding operations, which are challenging to implement efficiently in hardware due to their high memory and computational demands. Furthermore, the reliance on large expander graphs for encoding introduces significant memory constraints. In typical implementations, these graphs require considerable storage and memory bandwidth, which poses challenges for systems with limited resources. Another critical challenge is the generation of Merkle Trees for efficient proof commitments. Constructing these trees quickly while maintaining integrity and privacy guarantees adds further complexity to the proof generation process.

Given the performance bottlenecks in the commitment phase of Orion, there is a strong need to explore optimization strategies that can accelerate parts of the proof generation. This paper addresses two primary sub-operations of the commitment phase for optimization: (1) improving the efficiency of expander graph construction and usage in the encoding phase, and (2) optimizing the generation of Merkle Trees for secure commitment schemes. By focusing on these two core aspects, we aim to significantly reduce the computational overhead associated with Orion's commitment phase, thereby enhancing its scalability and practicality for real-world applications.

## 1.1 Contributions

We present a series of optimizations and design strategies aimed at improving the algorithm complexity and also the hardware friendlessness of Orion. Our contributions include:

- Hardware-friendly Construction of Expander Graphs: We propose an iterative approach to encoding that is more suitable for hardware platforms compared to traditional recursive methods. Moreover, we present novel and hardware-friendly

expander graph evaluation strategies. These measures reduce the memory footprint and allow for parallel processing, resulting in significant performance improvements.

- Memory-efficient On-the-Fly Graph Generation: To overcome the memory limitations associated with expander graphs, we introduce a method for dynamically generating graphs during the encoding process. This minimizes memory usage from several gigabytes to only a few 64-bit seeds therefore enabling more efficient use of limited hardware resources.
- Efficient Merkle Tree Generation: We develop an optimized method for constructing Merkle Trees, reducing the computational overhead in the commitment phase. We enhance the overall efficiency of the commitment phase by focusing on parallel hash computations and interleaving it with the tree construction time,
- Scalable Architecture for Accelerated Commitment: We propose a scalable architecture for a efficient commitment phase that integrates our optimizations for expander graphs and Merkle Trees, which leads to a two orders of magnitude speedup in performance compared to existing software-based implementations.

**Organization:** The remainder of the paper is structured as follows: In Section 2, we provide an overview of the necessary background, including graph theory, expander graphs, coding theory, and Merkle Tree commitments. In Section 3, we provide a comprehensive study of proof system, functional commitments, and the Orion scheme. Section 4 delves into the specific challenges and proposed solutions for efficient expander graph construction. In Section 5, we discuss the challenges associated with Merkle Tree generation and our optimized hardware-friendly approach. Section 6 presents our results, comparing our optimized hardware implementation to existing baselines, and Section 7 concludes the paper with a discussion on future work.

## 2 Background and Motivation

This section offers the essential background required to understand the contributions of this paper. To ensure the paper is self-contained, the background section is relatively extensive. For a more detailed explanation of Orion, including proofs and protocols, refer to the original Orion paper [XZS22].

### 2.1 Notation and Acronyms

Natural numbers and field elements are denoted using lowercase letters, e.g., d. Vectors are represented by  $\vec{v}$ , while matrices are indicated by bold capital letters, such as **M**. The element from the *i*-th row and *j*-th column is  $\mathbf{M}[i, j]$ , and the *i*-th column vector is  $\mathbf{M}[:, i]$ . To denote the size of a set S, we use the notation |S|. The implementation of Orion uses an extension field  $\mathbb{F}_{p^2}$ , e.g.,  $GF((2^{61}-1)^2)$  in the software developed by the authors [XZS22]. The elements of this extension field can be represented as a degree-1 polynomial a + bz where a and b are from the base field  $\mathbb{F}_p$ . It is not always necessary to use an extension field if the base field is sufficiently large. We use poly(x) to refer to a function upper-bounded by a polynomial in variable x with a constant degree.  $\operatorname{negl}(x)$  is to refer to a negligible function, i.e.,  $\operatorname{negl}(N) \leq \frac{1}{\operatorname{poly}(N)}$  for sufficiently large N.

## 2.2 Graphs and Expanders

Graphs in discrete mathematics are networks of points. Formally, a graph is a set of vertices and edges, where each edge is an unordered pair of vertices representing a connection. We denote this as G = (V, E) with V as the vertex set and E as the edge set. Both sets are usually finite, though not necessarily always [Big02]. The *degree* of a vertex is the number of edges connected to it [Hei03].



Figure 1: Unbalanced bipartite expanders.

#### 2.2.1 Expander graphs

In an expander graph, any small subset of vertices has a large proportion of its vertices connected to vertices outside the subset [SS94], i.e., the subset 'expands' to neighbours. Formally, a graph G = (V, E) satisfies the expansion property if, for some constants  $\delta > 0$  and m > 0, we have:

$$\forall S \subset V, |S| \le m \Rightarrow |\{y \in V \setminus S : \exists x \in S \text{ such that } (x, y) \in E\}| \ge \delta|S| \tag{1}$$

This means that for any subset S of at most m vertices, the number of distinct vertices outside S that are connected to at least one vertex in S is at least a  $\delta$ -fraction of the size of S [Spi96]. Thus, the number of neighbouring vertices grows proportionally with the size of the subset.

**Bipartite expanders:** The Orion proof system uses expander graphs that are also bipartite. In a bipartite graph, the set of vertices can be divided into two disjoint sets L and R such that there are no edges within L nor R. Thus, a vertex in L can only be connected to vertices in R and vice versa. We call L the left vertex set and R the right vertex set. We call such a bipartite graph G = (L, R, E) as (c, d)-regular if all vertices in L have the degree c and all vertices in R have the degree d. In addition, we define the expansion parameter  $\alpha$  such that  $|R| = \alpha |L|$ .

Given parameters  $\varepsilon$ ,  $\delta$  with  $0 \le \varepsilon < 1$  and  $\delta > 0$ , a (c, d)-regular graph is a  $(c, d, \varepsilon, \delta)$ expander if it upholds the expansion property mentioned in Equation 1 for an  $\varepsilon$ -fraction of the larger vertex set L or R. In Orion, we have |L| > |R|, thus we substitute V by L in Equation 1 and set  $m = \varepsilon |L|$ . The expansion property now tells us that for every subset of left vertices, there must be outgoing connections to R depending on  $\delta$ . Figure 1 illustrates such an expander. It connects a left vertex set of  $\{l_1, \ldots, l_k\}$  to a right-vertex set  $\{r_1, \ldots, r_n\}$  with dense connections. In the second subfigure, an example is given with the expansion of vertices  $\{l_2, l_3\}$  highlighted with c = 2 and d = 3.

### 2.2.2 Random expanders

Constructing expander graphs is typically challenging. However, using a random construction, where edges are randomly placed between vertices, can yield good results with relatively low complexity [Spi96]. Thus, using randomized edges is a convenient way to construct a bipartite (c, d)-regular expander. Let L be the left vertex set and R be the right. Then, we can construct the edge set as follows. Set  $E_l = \{(l, r) : r \text{ random vertex} \in R\}$ with  $|E_l| = c$  for each vertex  $l \in L$ . That is, for vertex l of L find c random vertices in Rto connect to. Then  $E = \bigcup_{l \in L} E_l$  the union of all random edge-sets allows to construct



Figure 2: Illustration of the recursion of Spielman codes

 $G = (L \cup R, E)$ . With high probability, the chosen edges result in d neighbours for all vertices in R [Spi96].

It is often desirable to prove that a particular graph G is a good expander. An efficient test for a good expander is provided in [XZS22]. They prove that a random construction as described above gives a good expander with probability  $1 - O\left(\frac{1}{\text{poly}(k)}\right)$ , where k = |L|. They define the Very Small Set Expansion problem distinguishing two cases [XZS22]:

- 1. Non-expanding:  $\exists S \subset L$  with  $|S| \leq \log \log k$  and Equation 1 does not hold for S.
- 2. Expanding:  $\forall S \subset L$  with  $|S| \leq \log \log k$ , Equation 1 holds.

To test for the non-expanding case, the authors introduce an algorithm given in Algorithm 2. If this algorithm outputs NotFound, then with overwhelming probability, the graph is a  $(c, d, \varepsilon, \delta)$ -expander.

## 2.3 The Spielman Linear Code

The goal of an error-correcting code is to detect and correct errors, e.g., transmission errors due to the unreliability of networks [HP10]. A code is defined over an alphabet, commonly consisting of binary digits. A message with length k is *encoded* into a *codeword*, which is represented using exactly n digits of the alphabet. Of the n digits, k digits are associated with the information, while the remaining m = n - k digits are used for error detection and correction. Such a code is referred to as an (n, k) code. The *rate* of the code is  $\frac{1}{r}$  with r = k/n, which is the ratio of digits used by the code against the minimum number of digits necessary to contain the same information.

Linear codes are the most studied form of error-correcting codes. Their characteristic is that any linear combination of codewords is again a valid codeword [RL09]. The work in [Spi96] proposes the Spielman code, a linear error-reduction code with a *linear-time encoding* using expander graphs. Orion [XZS22] uses this linear-time encoding to achieve a linear-time polynomial commitment scheme. We hence explain the generalization of linear Spielman codes as used in Orion, which works over finite fields.



Figure 3: Merkle tree example with l = 2, hash values for opening  $m_3$  marked in red

The construction of a linear Spielman code, denoted as  $E_C$ , for a message x uses a recursive encoding, as shown in Figure 2. The recursion step or level is indicated using superscript. At every level, two types of expander graphs  $G_1$  and  $G_2$  are used. In the first step, the k-field element-wide message x is encoded by the expander  $G_1^{(1)} = (L_1^{(1)}, R_1^{(1)}, E_1^{(1)})$  with  $|L_1^{(1)}| = k$  and  $|R_1^{(1)}| = \alpha_1 k$ ,  $0 < \alpha_1 < 1$ . This encoding results in  $m_1$  with  $\alpha_1 k$  elements. Then, this procedure is applied recursively to encode  $m_1$ , as shown in Figure 2. As soon as the recursion computes the codeword for  $m_1$  and returns to level-1, a different expander  $G_2^{(1)} = (L_2^{(1)}, R_2^{(1)}, E_2^{(1)})$  is applied to the codeword of  $m_1$  which results in the final  $c_1$  component of the codeword of x. For  $G_2^{(1)}$  we have  $|R_2^{(1)}| = \alpha_2 |L_2^{(1)}|$  with  $0 < \alpha_2 < 1$ . The final Spielman codeword of x is the concatenation of x, the codeword of  $m_1$ , and  $c_1$ , as shown at the top (level 1) of Figure 2.

All expanders  $G_1^{(i)}$  and  $G_2^{(i)}$  for recursion *i* use the constant parameter  $\alpha_1$  and  $\alpha_2$ , respectively. Since  $\alpha_1 < 1$ , the encodings  $m_i$  shrink in their size during the recursions, and at some point  $m_i$  will have less than  $n_0$  elements left, where  $n_0$  is a scheme-specific threshold for returning the recursion. Once this threshold is reached, no further recursion occurs, and the codeword is formed by appending the input message with the result of  $G_2^{(1)}$ , as in Figure 2. The expanders used in Orion can be randomly generated as discussed in Section 2.2.2.

The actual evaluation of an expander graph G = (L, R, E) with |L| = k and  $|R| = \alpha k$ over a k-element wide message can be expressed as matrix-vector multiplication. For that, the  $k \times \alpha k$  adjacency matrix **A** of the graph G is used. This matrix has  $\mathbf{A}[i][j] = 0$  if there is no edge connecting node i in L and node j in R. Conversely, if  $\mathbf{A}[i][j] = \omega_{i,j} \neq 0$ , there is such an edge, and a random weight  $\omega_{i,j}$  is assigned to the edge. Based on that, the graph evaluation result m and the message x are interpreted as row vectors  $\vec{m}$  and  $\vec{x}$ , whereby  $\vec{m} = \vec{x} \mathbf{A}$ .

### 2.4 Merkle Trees

A Merkle Tree is a binary tree, used to commit to a vector of  $2^l$  messages [Gol01] efficiently using a single hash value h at the root of the tree. The  $2^l$  leaf nodes store the cryptographic hashes of the  $2^l$  messages. Each non-leaf node stores the hash of its two children nodes. The root of the tree, h, serves as the final commitment, known as the Merkle commitment.

To prove the inclusion of any individual message  $m_i$  in the committed message vector, a Merkle proof  $\pi_i$  is generated. This proof consists of l hash values in a path to the root: starting from the leaf hash $(m_i)$  and the sibling node's hash at each level of the tree, all the way up to the root. A verifier combines these sibling hashes to re-compute the root hash and accepts  $\pi_i$  if and only if the re-computed root hash is h. The proof generation does not reveal any other messages  $m_j \forall j \neq i$ .

Figure 3 shows a representation of the Merkle Tree for a vector of four messages

 $[m_1, \ldots, m_4]$ . To prove that  $m_3$  is part of the committed vector, the prover provides the verifier with the siblings that appear in the path to the root, i.e.,  $\pi_3 = \{h_{2,4}, h_{1,1}\}$ . The verifier re-computes  $h_{2,3}$  from  $m_3$ , then  $h_{1,2}$  from  $h_{2,3}$  and  $h_{2,4}$ , and finally the root-hash from  $h_{1,1}$  and  $h_{1,2}$ . If the prover tries to cheat by claiming that a different  $m'_3$  is present in the committed vector, then the recomputed hash will not match with the commitment h with very high cryptographic probability. This proof size is logarithmic in the number of elements in the committed vector. Such logarithmic efficiency is particularly important for cryptographic proof systems.

## 3 Proof system

Proof systems are protocols that allow a prover to convince a verifier of the correctness of a statement. These proofs do not need to reveal any private information unless required by the protocol. For example, the prover can demonstrate knowledge of a private witness w for a public input x such that a circuit  $\mathcal{C}(x, w) = 0$  holds, without disclosing w. With this hiding feature, the proof system is called 'zero-knowledge'.

Proof systems are commonly used to ensure the correctness of computations outsourced to untrusted entities. The verifier can verify the proof using a small amount of computation, hence efficiently, without performing the outsourced computation themselves. There are several important performance metrics for proof systems, such as the proof generation time, proof size, and proof verification time. Usually, proof systems are optimized for succinct proofs and low verification costs. Having a low proof generation time is a plus and desired in applications that require scalability.

Modern succinct non-interactive proof systems are built by combining a functional commitment scheme with an interactive oracle proof (IOP) to form a succinct interactive argument, then applying the Fiat-Shamir transform [FS86] to make it non-interactive. In the following part, we describe functional commitment with a focus on the specific type 'polynomial commitment'.

### **3.1** Functional commitment (with a focus on polynomial commitment)

A functional commitment scheme is a commitment scheme that enables a party to commit to a function (computation procedure) they intend to evaluate. A prover can prove statements about the committed function, e.g., the evaluation of the function at a given point is correct. Functional commitment schemes have binding and optionally hiding properties. The biding property ensures that the committer cannot change the function or the input to the function so that the commitment remains unaffected. The hiding property, which is required for *zero-knowledge* succinct non-interactive argument of knowledge or SNARK construction, ensures that no information about the function or the inputs to the function is revealed. A functional commitment scheme is a cryptographic protocol with an underlying cryptographic assumption, e.g., discrete logarithm assumption, pairing assumption, cryptographic hash assumption, etc.

The Merkle Tree presented in the previous subsection is a vector commitment scheme to prove membership in a committed vector. While vector commitments like Merkle Trees are useful for committing to a static set of values and proving membership, they lack the flexibility, efficiency, and built-in polynomial structure needed for polynomial-related operations. Polynomial commitments [KZG10] are functional commitments, specifically designed to address these limitations by enabling succinct proofs of evaluations, degree checks, and functional properties of polynomials, making them essential in many cryptographic applications. With a polynomial commitment scheme, a prover can commit to a polynomial, for example,  $\phi(x) \in \mathbb{F}[x]$  of degree t with coefficients from the field  $\mathbb{F}$ . Simply committing to each coefficient  $(\phi_0 \| \dots \| \phi_t)$  of the polynomial is ineffective as verifying the commitment would require revealing the entire polynomial. Furthermore, many cryptographic applications require that only evaluations of  $\phi(x)$  at specific x are revealed without revealing the entire polynomial. Polynomial commitments play a crucial role in modern proof systems e.g., SNARKs.

A polynomial commitment scheme [KZG10] consists of four main steps. First, the 'Setup Phase' involves the generation of public parameters using the SETUP algorithm for a given function family, say  $\mathcal{F}$ . In the second step, the 'Commitment Phase', the prover P uses the COMMIT algorithm and the public parameters to generate a commitment  $com_{\phi}$  for a polynomial  $\phi \in \mathcal{F}$ . Third, during the 'Evaluation Phase', the verifier  $\mathbf{V}$  selects an evaluation point x = a and requests the prover to evaluate  $\phi$  at that point. The prover computes the value  $b = \phi(a)$  and generates an evaluation proof  $\pi$  that asserts the correctness of this evaluation. Finally, in the 'Verification Phase', the verifier checks the validity of the proof  $\pi$  against the commitment  $com_{\phi}$ , the evaluation point a, and the claimed output b, accepting or rejecting the claim based on the result.

**Example polynomial commitment:** To explain how a polynomial commitment scheme works, we use the famous KZG scheme [KZG10] with some simplifications. Let  $\phi(x) = \phi_0 + \phi_1 x + \cdots + \phi_t x^t$  be a polynomial over  $\mathbb{Z}_p$  of degree t. The scheme uses the discrete logarithm and bilinear pairing cryptographic assumptions and requires a trusted setup. The setup generates public parameters  $PP = (g, g^s, g^{s^2}, \ldots, g^{s^t}, g_2, g_2^s)$  where  $g \in G_1$  and  $g_2 \in G_2$  are generators of two bilinear groups of prime order p, and s is a secret 'toxic waste' destroyed after the setup. The bilinear pairing  $e: G_1 \times G_2 \to G_T$  is used for verification. To commit to  $\phi(x)$ , the prover computes  $com_{\phi} = g^{\phi(s)} = \prod_{i=0}^t (g^{s^i})^{\phi_i}$  which is an element of  $G_1$ . To prove that the evaluation  $y_a = \phi(x_a)$  at  $x = x_a$  is correct, the prover computes the quotient polynomial  $q(x) = \frac{\phi(x)-y_a}{x-x_a}$  and then the proof  $\pi = g^{q(s)} \in G_1$ . The verifier checks the evaluation proof  $\pi$  by using the bilinear pairing  $e(com_{\phi}/g^{y_a}, g_2) \stackrel{?}{=} e(\pi, g_2^{s-x_a})$  where  $g_2^{s^{-x_a}}$  is precomputed from the public parameters.

The prover in KZG polynomial commitment scheme is very slow and due to the use of elliptic curve cryptography, the scheme is not quantum-resilient. Furthermore, the scheme requires a trusted setup. In the following, we present the basics of Orion [XZS22] which provides fast proving times, quantum resilience, and non-trusted setup.

## 3.2 Orion proof system

Orion [XZS22] is a highly efficient zero-knowledge proof system that utilizes linear-time polynomial commitments and Merkle Trees to achieve succinct proofs with fast proving and verification. Unlike most contemporary systems with superlinear prover times [KZG10, Gro16, CHM<sup>+</sup>20], Orion's prover runs in linear time. It only relies on hash functions, making it resistant to quantum attacks, unlike schemes based on discrete logarithm or pairing assumptions. Built on linear codes via expander graphs, Orion uses code-switching to achieve sublinear proof sizes.

Consider a multilinear polynomial  $\phi$  in log N variables where each variable has a degree 0 or 1. There are N monomials and coefficients in  $\phi$ . The coefficients of  $\phi$  are in a field, say  $\mathbb{F}_{p^2}$ , and any evaluation of  $\phi$  is also performed in  $\mathbb{F}_{p^2}$ . Authors in [GLS<sup>+</sup>21] found that such a polynomial evaluation can be expressed as a tensor product. The evaluation of  $\phi$  at  $\vec{x} = [x_0, \ldots, x_{\log N-1}]$  can be written as

$$\phi(\vec{x}) = \sum_{i_0=0}^{1} \dots \sum_{i_{\log N-1}=0}^{1} w_{i_0,\dots,i_{\log N-1}} x_0^{i_0} x_1^{i_1} \dots x_{\log N-1}^{i_{\log N-1}}$$
(2)

In the above expression,  $i_0, \ldots, i_{\log N-1}$  represent the binary decomposition of monomial index  $i \in \{0, 1, \ldots, N-1\}$ , and the  $w_i$  terms correspond to the coefficients of the monomials

**Protocol 1** Overview of Orion [XZS22] with simplifications

**Public input:** Evaluation point  $\vec{x}$  parsed as tensors  $\vec{r_0}$  and  $\vec{r_1}$ ;

**Private input:** Polynomial  $\phi$  with coefficients  $\vec{w}$ ; Let  $E_C$  be the encoding function of a [n, k, d] linear code and  $N = k^2$ ;

- 1: function  $COMMIT(\phi)$
- 2: Parse the coefficient vector  $\vec{w}$  of length N as a  $k \times k$ -matrix **W**;
- 3: Using  $E_C$  encode each row of **W** to obtain code **C** which is a  $k \times n$  matrix;
- 4: for  $0 \le i < n$  do
- 5: Compute Merkle root for each column  $Root_i \leftarrow Merkle.Commit(\mathbf{C}[:,i]);$
- 6: Compute the Merkle root  $\mathcal{R} \leftarrow Merkle.Commit; ([Root_0, \dots, Root_{n-1}])$
- 7: Output  $\mathcal{R}$  as the commitment;
- 8: function PROVE $(\phi, \vec{x}, \mathcal{R})$
- 9: Prover receives from the verifier a random vector  $\vec{\gamma}_0 \in \mathbb{F}_{p^2}^k$ ;
- 10:  $\vec{c}_{\gamma_0} \leftarrow \sum_{i=0}^{k-1} \vec{\gamma}_0[i] \mathbf{C}[i,:], \ \vec{y}_{\gamma_0} \leftarrow \sum_{i=0}^{k-1} \vec{\gamma}_0[i] \mathbf{W}[i,:]; \qquad \triangleright \text{ Proximity check}$ 11:  $\vec{c}_1 \leftarrow \sum_{i=0}^{k-1} \vec{r}_0[i] \mathbf{C}[i,:], \ \vec{y}_1 \leftarrow \sum_{i=0}^{k-1} \vec{r}_0[i] \mathbf{W}[i,:]; \qquad \triangleright \text{ Consistency check}$ 12: Prover sends the evaluation of  $\phi$  at  $\vec{x}$  as  $y \leftarrow \langle \vec{y}_1, \vec{r}_1 \rangle$  to verifier;
- /\* In the following, the prover's messages to the verifier form the proof string  $\pi_{\vec{x}}$  \*/
- 13: Prover sends  $\vec{c}_{\gamma_0}$ ,  $\vec{y}_{\gamma_0}$ ,  $\vec{c}_1$ , and  $\vec{y}_1$  to the verifier;
- 14. Verifier sends the set  $\hat{I}$  of t (0 < t < n) randomly sampled column indexes;
- 15: **for**  $idx \in I$  **do**
- 16: Prover sends to verifier  $\mathbf{C}[:, idx]$  with Merkle proof of  $\mathtt{Root}_{idx}$  under  $\mathcal{R}$ ;
- 17: function VERIFYEVAL $(\mathcal{R}, \vec{x}, y = \phi(\vec{x}), \pi_{\vec{x}})$

/\* Verifier parses the proof string  $\pi_{\vec{x}}$  and obtains the prover's messages \*/

18:  $\langle \vec{y}_1, \vec{r}_1 \rangle == y;$ 

19:  $E_C(\vec{y}_{\gamma_0}) == \vec{c}_{\gamma_0} \text{ and } \forall \operatorname{idx} \in \hat{I}, \langle \vec{\gamma}_0, \mathbf{C}[:, \operatorname{idx}] \rangle == \vec{c}_{\gamma_0}[\operatorname{idx}]; \qquad \triangleright \operatorname{Proximity check}$ 20:  $E_C(\vec{y}_1) == \vec{c}_1 \text{ and } \forall \operatorname{idx} \in \hat{I}, \langle \vec{r}_0, \mathbf{C}[:, \operatorname{idx}] \rangle == \vec{c}_1[\operatorname{idx}]; \qquad \triangleright \operatorname{Consistency check}$ 

- 21: Check  $\forall idx \in \hat{I}, \mathbf{C}[:, idx]$  is consistent with  $\mathtt{Root}_i$  and its Merkle Tree root  $\mathcal{R}$ ;
- 22: Accept if all checks pass;

 $X_i = x_0^{i_0} x_1^{i_1} \dots x_{\log N-1}^{i_{\log N-1}}$ . Let  $\vec{w}$  denote the vector of monomial coefficients with the *i*-th element  $\vec{w}[i] = w_{i_0,\dots,i_{\log N-1}}$ . Assuming N has an integer square root (e.g., N a power of 2), let  $k = \sqrt{N}$ . Also, let  $\vec{r}_0 = (X_0, X_1, \dots, X_{k-1})$  and  $\vec{r}_1 = (X_{0\cdot k}, X_{1\cdot k}, \dots, X_{(k-1)\cdot k})$  be two vectors containing k distinct monomials. Then all the monomials in Equation 2 are obtained using the tensor product  $\vec{r}_0 \otimes \vec{r}_1$ . Finally, the evaluation of  $\phi$  can be obtained using the inner product  $[\text{GLS}^+21]$  as  $\phi(x_0,\dots,x_{\log N-1}) = \langle \vec{w}, \vec{r}_0 \otimes \vec{r}_1 \rangle$ . Protocol 1 gives an overview of Orion's polynomial commitment scheme [XZS22] with some simplifications.

**Commitment:** In line-2 of the protocol, the coefficient-vector  $\vec{w}$  of length  $N = k^2$  is parsed as the matrix  $\mathbf{W}$  of dimension  $k \times k$ . Note that the evaluation  $y = \phi(\vec{x}) = \langle \vec{w}, \vec{r}_0 \otimes \vec{r}_1 \rangle$  can also be represented as a vector-matrix-vector multiplication as follows: the row-vector  $\vec{r}_0$ is multiplied from the left with matrix  $\mathbf{W}$ , and the resulting row-vector is multiplied with the column-vector  $\vec{r}_1$  to produce y. Orion utilizes the tensor IOP protocol from [BCG20] to construct a polynomial commitment based on [GLS<sup>+</sup>23].

Let  $E_C$  be the encoding procedure of an [n, k, d] linear code (Section 2.3). In line 3, each row  $\mathbf{W}[i, :]$  is encoded into a codeword of length n. After encoding all rows of  $\mathbf{W}$ , we obtain the code matrix  $\mathbf{C}$  of dimension  $k \times n$ . In lines 4 and 5, the columns of  $\mathbf{C}$  are Merkle-committed to leaf hashes  $\text{Root}_i$ . Finally, from the n leaf hashes, the Merkle Tree is

calculated in line 6 to produce the root hash as the final commitment to the tree. Orion uses the linear-time Spielman code (described in the previous section) to achieve linear time complexity for the polynomial commitment.

**Proving mechanism:** The prover and verifier engage in an interactive protocol starting from line 8 in Protocol 1. As a challenge, the verifier provides the prover with the random vector  $\vec{\gamma}_0 \in \mathbb{F}_{p^2}^k$ . Using that, in line 10 of the protocol, the prover computes random linear combinations of the rows of **C** and **W**. As the code is linear, any linear combination of codewords is also a codeword. Furthermore, if the calculations are performed correctly (i.e., non-cheating prover), the resulting codeword  $\vec{c}_{\gamma_0}$  should be the encoding of  $\vec{y}_{\gamma_0}$ . Next, in line 11, a similar linear combination is performed using  $\vec{r}_0$  of the tensor query  $\vec{r}_0 \otimes \vec{r}_1$ . If performed correctly,  $\vec{c}_1$  will be the encoding of  $\vec{y}_1$ . Furthermore, given  $\vec{y}_1$ , the verifier will be able to re-compute y using the inner product  $\langle \vec{y}_1, \vec{r}_1 \rangle$  spending  $\sqrt{N}$  effort.

The verifier performs two important checks, namely the proximity and consistency check. The verifier randomly samples the set  $\hat{I}$  containing t column-indexes of **C**. During the checks, the prover sends the specified columns along with their Merkle proofs under the commitment  $\mathcal{R}$ .

- **Proximity check:** The proximity check in line 19 ensures that the prover's encoded messages (whereas the messages are structured coefficients of the polynomial  $\phi$ ) are close to valid codewords, confirming that the prover's inputs lie within the correct space, e.g., low-degree polynomials. Due to the properties of linear codes, a linear combination of codewords is again a codeword. Thus,  $\vec{c}_{\gamma 0}$  must be a codeword for the message  $\vec{y}_{\gamma 0}$ . Furthermore, to check that  $\vec{c}_{\gamma 0}$  was generated genuinely by combining  $\vec{\gamma}_0$  with the committed code matrix **C**, the verifier checks the inner products between  $\langle \vec{\gamma}_0, \mathbf{C}[:, \mathrm{idx}] \rangle == \vec{c}_{\gamma 0}[\mathrm{idx}]$  at all random indices in the set  $\hat{I}$ . As the columns of **C** are Merkle-committed under  $\mathcal{R}$ , the prover cannot cheat by sending false columns. The size t of the index set  $\hat{I}$  is sufficiently large to ensure a negligible probability with which a cheating prover can pass.
- Consistency check: The goal of the consistency check in Orion is to ensure that the prover's claims and intermediate computations or evaluations are coherent and match with the commitment. Thus, it prevents the prover from altering values during the protocol. The consistency performs very similar computations like the proximity check but involves the  $\vec{r}_0$  component of the tensor query to compute the linear combinations. It is enough for the prover to convince that it computed  $\vec{y}_1$ correctly. The verifier can check the correctness of the final result y by recomputing  $y == \langle \vec{y}_1, \vec{r}_1 \rangle$ . As  $\vec{y}_1$  and  $\vec{c}_1$  are computed as linear combinations using the same  $\vec{r}_0$ , for a genuine prover  $\vec{c}_1$  must be the encoding of  $\vec{y}_1$ . Additionally, to verify that  $\vec{c}_1$ was correctly computed by combining  $\vec{r}_0$  with the committed code matrix  $\mathbf{C}$ , the verifier checks the inner products  $\langle \vec{r}_0, \mathbf{C}[:, idx] \rangle == \vec{c}_1[idx]$  at random indices from the set  $\hat{I}$ . Since the columns of  $\mathbf{C}$  are Merkle-committed under  $\mathcal{R}$ , the prover cannot cheat by submitting incorrect columns.

Both checks rely on the constant relative distance of the linear code, which guarantees that the committed matrix is close to a tensor codeword, and if so,  $y = \phi(\vec{x})$  is a valid evaluation with overwhelming probability.

Orion has  $\mathcal{O}(N)$  time commitment and proof generation. However, the proof size in Protocol 1 is  $(\sqrt{N})$ , which is quite large compared to commonly used pairing-based proof systems. To overcome this limitation, the authors of Orion [XZS22] use a so-called 'proof composition' technique [RZR24] and reduces the size to  $\mathcal{O}(\log^2 N)$ .

**Generalization to**  $N = k_1 \cdot k_2$  For simplicity, the above-mentioned description of encoding and Orion used  $N = k^2$ , resulting in **W** of dimension  $k \times k$ . Assuming N is a power-of-two,

Size	Latency o	Total (in ms)			
$\log(N)$	Initialize	Commitment	<b>Open+Verify</b>		
16	1	40	50	92	
18	6	163	127	296	
20	24	700	399	$1,\!124$	
22	115	$3,\!175$	1,503	4,793	
24	670	13,756	6,261	$20,\!687$	
26	$3,\!837$	60,011	$27,\!240$	91,088	
28	$18,\!614$	$334,\!250$	$118,\!100$	470,964	

Table 1: Timing result of different operations in reference implementation of Orion [Ori]; Results collected in software on an AMD EPYC 9754 @2.25GHz.

Table 2: Reference timings of standard Expander Graphs in seconds; Results collected in software on an AMD EPYC 9754 @2.25GHz.

Size	Total (in ms)				
$\log(N)$	Initialize	$\mathrm{Encode}(\mathit{msg})$	$\operatorname{Hash}(C)$	$\operatorname{MerkleTree}(H)$	
16	4	25	10	1	40
18	10	104	40	1	164
20	73	463	163	2	801
22	279	2.233	656	7	$3,\!175$
24	1,998	10.080	$2,\!651$	27	14,756
26	$3,\!448$	45,855	$10,\!600$	108	60,011
28	$13,\!137$	278,100	42,580	433	$334,\!250$

it is easy to see that the same can be generalized to asymmetric decomposition  $N = k_1 \cdot k_2$ . The software implementation of Orion [Ori] by the authors uses the fixed  $k_1 = 128$  for all large N. Depending on the value of N, the number of columns  $k_2$  in **W** is adjusted. This gives flexibility. Following the authors of Orion, we use fixed  $k_1 = 128$  in our hardware implementation.

### 3.3 Latency Operations Performed in Orion

In this section, we present a detailed latency analysis of Orion's core operations, particularly how the system's performance scales as N increases. Note that in contrast to cryptographic schemes, such as the one-time key generation process in digital signatures, the commitment phase in Orion is repeatedly invoked as part of a recursive proving system. E.g. When creating proofs of Training for Deep Neural Networks as in [APKP24] proofs are required for each layer. Each time the recursive prover processes a new layer or polynomial, it calls Orion to commit to the new data. As a result, the commitment phase must efficiently handle repeated polynomial commitments across multiple invocations, making it computationally intensive.

Orion's execution can be broken down into three main phases: Initialization, Commitment, and Open + Verify. While all phases contribute to overall latency, the commitment phase stands out due to its computational complexity, particularly in managing the polynomial commitments across recursive calls.

In the software [APKP24] of Orion the Initialization Phase plays a crucial role by constructing expander graphs, which are essential for establishing the system's foundational structure. However, as shown in Table 1, its execution time remains relatively constant and manageable across different values of N. In contrast, the Open + Verify phase

contributes more significantly to the system's latency, with its impact increasing as N grows. However, the Commitment Phase stands out as the most time-intensive operation in Orion. This phase consists of several computationally demanding sub-processes, including linear encoding, hashing, and Merkle Tree generation. The latency of the commitment phase is considerably higher compared to the other phases as shown in Table 1. This makes it the dominant factor in the total execution time, especially as N increases. These results suggest that the commitment phase is both resource-intensive and the most suitable operation for acceleration.

Table 2 gives a breakdown of the sub-processes within the commitment phase, mainly linear encoding, hashing, and Merkle Tree generation. The most prominent operation is linear encoding, which takes the input and encodes it by using an expander graph. The encoding process as well as expander graphs are described in detail in Section 2.3. Following the encoding, the hashing of the columns of the encoded matrix **C** for Merkle Tree construction represents the second largest time cost. Although the hashing process leverages hardware intrinsics for the SHA functionality in the software implementation [Ori], the sheer amount of data involved makes it less efficient.

## 4 Accelerating Linear Encoding through Inverted Expander

The efficiency of the Orion protocol is mainly achieved by the linear encoding process using expander graphs. These graphs are carefully designed to be sparse, with a limited number of edges per node, while maintaining high connectivity to preserve the security and succinctness required for cryptographic protocols. This dual property is essential because the sparsity ensures efficient computations, while high connectivity guarantees soundness and completeness of the encoding. However, balancing these two properties is non-trivial. The straightforward process of constructing and using expander graphs involves Gigabytes of memory and random accesses to this large memory. These random accesses substantially lower the memory bandwidth [Xil24]. In addition, frequent read and write interleavings, as used in baseline Orion [Ori], cause significant turnaround overheads in the DRAM memory system [CKLE20]. Therefore, a careful and holistic hardware accelerator design is essential to cope with these hazards and achieve high performance.

The remainder of this section explains our novel optimizations for efficient and hardwarefriendly linear encoding using expander graphs. We detail our iterative encoding procedure before proposing our graph inversion technique, which significantly reduces the random memory accesses and read-write turnarounds. Based on this technique, we finally present the corresponding hardware architecture for linear encoding.

## 4.1 Iterative linear encoding

The linear encoding of data is a pivotal factor for the performance of the Orion protocol, particularly in the commitment phase. Orion encodes the coefficient matrix  $\mathbf{W}$  of a polynomial into a code matrix  $\mathbf{C}$  in a row-wise manner. In essence, each row in  $\mathbf{W}$  is encoded into one row on  $\mathbf{C}$  as detailed in Section 3.2. In the software implementation of Orion, the linear encoding algorithm for one row in  $\mathbf{W}$  is recursive, as detailed in Section 2.3. However, a recursive approach is not ideal for hardware designs due to the increased control overhead. Thus, we present an iterative approach for linear encoding, as presented in Figure 4.

The iterative linear encoding procedure starts with applying an expander graph  $G_1^{(1)}$  with expansion parameter  $\alpha_1$  to the input message, which is one row of **W** (Figure 4 top). The result of this operation is the encoding  $m_1$ , which is smaller in size than the input message. In the next step, another expander graph  $G_1^{(2)}$  with  $\alpha_1$  is applied on  $m_1$  yielding



Figure 4: Our iterative approach for computing Spielman codes.

 $m_2$ . Thereby,  $m_2$  is again smaller than  $m_1$ . This procedure is applied until the size of the output encodings  $m_i$  is below a certain threshold  $n_0$ .

Thereafter, different expander graphs  $G_2^{(j)}$  with expansion parameter  $\alpha_2$  are applied iteratively on the previous codewords, as shown in Figure 4. The results of the  $\alpha_2$  expanders are denoted with  $c_j$ . The graph evaluation with parameter  $\alpha_2$  is repeated until all previous codewords are consumed. The final expander graph yields  $c_1$ , which completes the linear encoding. This iterative encoding approach heavily relies on expander graph evaluation, which is the critical operation. Hence, we optimize the expander graph evaluation and present our efficient inverted expander graphs in the next section.

### 4.2 Optimized Memory Access Pattern through Graph Inversion

The evaluation of expander graphs is the key operation in Orion's linear encoding procedure. The expander graph implementation in the baseline Orion software is tailored for software platforms but introduces substantial overheads in hardware due to random read and write accesses to off-chip memory. The off-chip memory such as HBM is required due to the large amount of data involved. At the same time, the memory bandwidth of HBM degrades through random accesses by more than  $3 \times$  compared to linear accesses [Hub19]. Furthermore, frequent read-write turnarounds also introduce substantial latencies [CKLE20].

To address these challenges of Orion's baseline linear encoding, we propose an *inverted* expander graph which significantly reduces random read and write accesses and minimizes read-write turnarounds. This allows our methodology to enhance the achievable performance. In this section, we first explain the straightforward expander graph implementation in Orion and show the accompanying disadvantages of that approach for hardware designs. Then, we present our inverted expander graph method to improve the memory bandwidth.

#### 4.2.1 Baseline Expander Graph Computation

The expander graph evaluation used in Orion maps some input message x to the output m. This can be expressed as matrix-vector multiplication, as discussed in Section 2.3. Although the matrix-vector multiplication is illustrative, using this approach for actual implementations is impractical. This is explained by the sparse adjacency matrix which mostly consists of zeros. A more practical approach to evaluating expander graphs



Figure 5: The two different approaches of expander graph implementation. Only a few weights  $\omega_{i,t}$  are explicitly shown, but every edge connecting  $l_i$  to  $r_t$  has a weight  $\omega_{i,t}$ .

G = (L, R, E) is explained in Figure 5a. Thereby, the left node set  $L = \{l_0, l_1, \ldots\}$  and the right node set  $R = \{r_0, r_1, \ldots\}$  are interpreted as arrays of field elements  $l_i, r_t \in \mathbb{F}_{p^2}$ . In baseline Orion, the left (input) nodes  $l_i$  in the expander graph have a constant degree c (c = 3 in this example). In contrast, the degree of the right nodes  $r_t$  varies due to the random assignment of left nodes to right nodes (Section 2.2.2). This causes a normal distribution of the number of incoming nodes at  $r_t$  with a certain mean and variance. In addition, each edge connecting  $l_i$  to  $r_t$  has a randomly assigned weight  $\omega_{i,t} \in \mathbb{F}_{p^2}$ .

For evaluating the expander graph, the baseline scheme initializes all  $r_t \leftarrow 0$  and iterates linearity over  $l_i$ , starting with i = 0. In the first iteration, the value  $l_0$  is multiplied with c weights  $\omega_{0,t_1} \dots \omega_{0,t_c}$ . The c multiplications are then accumulated to  $r_{t_1} \dots r_{t_c}$ , according to the expander graph. Consider the example marked red in Figure 5a. In this example, the value  $l_0$  is multiplied by c = 3 weights and accumulated to  $r_0$ ,  $r_2$ , and  $r_3$  as in:

$$r_0 = r_0 + l_0 \cdot \omega_{0,0} \qquad r_2 = r_2 + l_0 \cdot \omega_{0,2} \qquad r_3 = r_3 + l_0 \cdot \omega_{0,3}. \tag{3}$$

Next, the execution advances to  $l_1$  and repeats the same steps as in  $l_0$ . This is done until all input nodes are handled making the output nodes  $r_t$  holding the encoding result m.

This straightforward approach requires random read and write operations, which lowers the effective memory bandwidth. For each input node  $l_i$ , c random right-nodes  $r_{t_1}$  to  $r_{t_c}$ must be read from off-chip memory, as storing the large data structure on the chip is infeasible. Then, the c accumulations are performed (as in Equation 3) and the c results are stored back to scattered memory locations. This shows that, per input node, c random reads and c random writes are required. These random memory accesses in combination with frequent read/write turnarounds reduce the achievable memory bandwidth on HBM RAM drastically.

In addition to random memory accesses, the baseline approach introduces read-afterwrite hazards in the multiplication and accumulation pipeline. Consider two consecutive input nodes  $l_i$  and  $l_{i+1}$  mapping to the same output node  $r_t$ . Then, the computation of the prior node might still be executed in the pipeline whereas the computation of the latter node needs to be delayed. These issues are straightforward when computation is performed entirely on-chip but become more challenging when random-access and off-chip memory, such as HBM, is involved. This causes pipeline stalls and control logic overhead, thereby limiting the performance.

#### 4.2.2 Alternatives to Baseline Expander Graphs

We aim to improve the baseline expander graph computation as used in Orion and present three alternative graph computation methods. These methods are called Inverted method, Normal method, and Without Replacement method:

- **Inverted:** This method mirrors the process in Algorithm 3, which generates a constant number of c edges per left node to right nodes. Conversely, the Inverted method generates a constant number of d edges per right node to left nodes. Compared to the baseline expander graph, where the right nodes' degree d follows a normal distribution, Inverted graphs have a normally distributed left-side degree c, as shown in Figure 6a to 6c.
- **Normal:** In the Normal method, we neither fix c nor d. Instead, we randomly draw the degree d for each right-side node from a normal distribution. Then, we establish d many connections from the specific right-side node to randomly chosen left-side nodes. This approach leads to normally distributed c and d across the nodes, as shown in Figure 6d to 6f.
- Without Replacement: This algorithm aims for a fixed degree c for left-side nodes. This is achieved using the Normal approach and adapting it such that each left-side node can at most be drawn c' times. Hence, most of the left nodes have a degree c = c'. Yet, due to the randomly sampled d (see Normal approach), some left-side nodes may have less degree c < c'. The resulting distributions can be seen in Figure 6g to 6i. Figure 6g clearly shows that most left nodes have degree c = c' whereas a few have less degree.

We implement these three graph evaluation methods in software and parametrize the graph generation. This parametrization uses statistical analysis based on the baseline expander graph. A brief explanation of the statistical analysis can be found in Appendix C. After finding proper parameters, we examine the graphs' suitability in terms of expansion. For that, we rely on Algorithm 2 initially proposed in [XZS22]. This algorithm accepts our Inverted, Normal, and Without Replacement graphs with the same high probability as the baseline expander graphs. This indicates that our three approaches are valid alternatives to baseline expander graphs.

Method selection: For our hardware design, we choose the Inverted method. In contrast to the Normal and Without Replacement methods, the Inverted method does not need random sampling from a Gaussian distribution. This is beneficial for hardware designs since Gaussian sampling increases the complexity of the implementation. Instead, the Inverted method only requires uniform sampling which is hardware friendly. Based on this discussion, we elaborate on our Inverted expander graphs in the next section and show the efficiency gain achieved by this methodology.

### 4.2.3 Proposed Inverted Expander Graph Computation

In this section, we present *inverted expander graphs*, which address the limitations of Orion's baseline expander graphs discussed above. Unlike the baseline case, we fix the degree d in the *right* nodes  $r_t$  of the inverted expander graph, as shown in Figure 5b. Moreover, we relax the degree of the left nodes  $l_i$ , which should follow a Gaussian distribution instead of having a fixed value. This essentially mirrors the baseline expander graph, which had a fixed degree for  $l_i$  and Gaussian distributed degrees for  $r_t$ .

The resulting inverted expander graph allows us to iterate linearly over nodes  $r_t$ , starting from t = 0, as shown right in Figure 5b. For each  $r_t$ , d random nodes  $l_i$  are selected and multiplied with random weights. Finally, the multiplication results are added to give the final value of  $r_t$ .



Figure 6: Distributions of left and right node degree c and d for expander graphs using our three methods. Parameters for Orion's  $G_1^{(i)}$  graph.

Our inverted expander graph approach limits memory accesses to random reads of the left-side nodes  $l_i$  during computation, offering a significant advantage over baseline expander graphs, which require interleaved random reads and writes of the right-side nodes. As a result, we nearly halve the number of random memory accesses. Additionally, by minimizing read-write turnarounds, our method optimizes memory bandwidth utilization, particularly for off-chip memory like HBM. Lastly, our approach resolves read-after-write hazards, enabling the pipeline design in the architecture.

**Parameter selection:** Although our inverted expander graph approach allows more performant hardware designs, we must ensure that it preserves the desired expansion properties needed in the ZKP. We hence search for proper parameters  $d_1$  and  $d_2$  for the expander graphs  $G_1^{(i)}$  and  $G_2^{(i)}$ , respectively. Here,  $d_1$  and  $d_2$  are the degrees of the right-side nodes in the graphs (i.e. the number of incoming edges to the output nodes  $r_t$ ). Note, that in general,  $d_1$  and  $d_2$  in the inverted expander graphs differ from  $c_1$  and  $c_2$  in the baseline expander graph.

As a first approach, we use the baseline expander graph with a fixed degree c of left nodes  $l_i$  to experimentally obtain the expected degree E(d) of right nodes  $r_t$ . We take the found expected value as the degree of all right nodes in our inverted expander graph. This gives us the parameters  $d_1 = 42$  and  $d_2 = 26$  for  $G_1^{(i)}$  and  $G_2^{(i)}$ , respectively. In



Figure 7: Duration of linear encoding in software for different degrees  $d_1$  and  $d_2$ .

the next step, we generate inverted expander graphs and validate their suitability using Algorithm 2, which is also used by [XZS22] to validate their parameters. The validation algorithm accepted 99% of the randomly created baseline expander graphs and 99% of our inverted expander graphs. This shows the validity of our inverted expander graph parameters since we establish the same success probability of 0.99 as in [XZS22].

After finding an initial guess of parameters  $d_1$  and  $d_2$ , we fine-tune these parameters. Thereby, lower values for  $d_1$  and  $d_2$  lead to lower runtime. This is clearly visible in Figure 7, showing lower runtimes for smaller  $\{d_1, d_2\}$ , especially for large polynomial sizes. The reason for that is the lower amount of operations and fewer memory accesses caused by smaller parameters. Yet, smaller parameters lead to less expansion in the linear encoding. This is shown in Table 3, wherein smaller parameters tend to have a lower probability of sufficient expansion. An interesting observation from Table 3 is, that parameters smaller than  $d_1 = 42$ ,  $d_2 = 26$  still provide the same high probability of 0.99. One example for that is  $d_1 = 34$ ,  $d_2 = 22$ . Having smaller parameters results in lower runtime, as highlighted in Figure 7. We select the parameters  $d_1 = 34$  and  $d_2 = 22$  as default for our design. This choice leads to the lowest runtime among the results in Table 3. At the same time, the chosen parameters also ensure the expansion property with the same high probability as in the software implementation of Orion. Importantly, our hardware allows to customize the tradeoff between computation time and expansion probability by supporting runtime-configurable  $\{d_1, d_2\}$  thus offering flexibility. Yet, users must ensure to provide valid parameters  $\{d_1, d_2\}$  since too small parameters might fail in generating good expander graphs causing low minimal distance between codewords.

**Ensuring the expansion property:** Our inverted expander graph generation approach maintains a sufficient probability of valid graph generation verified through Algorithm 2 from [XZS22]. This verification ensures that the graph maintains its expander properties and performs consistently in the encoding process.

### 4.2.4 On-the-Fly Graph Generation

In the baseline Orion implementation, the entire expander graph is first generated and stored in memory before its usage. This procedure is shown in Algorithm 3 in the Appendix

Graph Type		Our Inverted Graph						[XZS22]			
$egin{array}{c} d_1 \ d_2 \end{array}$	$\begin{vmatrix} 34\\22 \end{vmatrix}$	$\begin{array}{c} 42 \\ 22 \end{array}$	$42^{*}$ 26*	$\frac{22}{22}$	$\frac{34}{26}$	$22 \\ 26$	$     34 \\     20   $	$\begin{array}{c} 42 \\ 20 \end{array}$	$22 \\ 20$	$\frac{16}{16}$	$c_1 = 10$ $c_2 = 20$
Probability	0.99	0.99	0.99	0.90	0.98	0.89	0.97	0.97	0.92	0.88	0.99

Table 3: Probability to sample a valid expander graph given different node degrees for our inverted expander graph and the baseline expander graph from [XZS22].

\*parameters found by analyzing the baseline graph

Algorithm 1 Our on-the-fly Inverted Expander Graph Computation							
1:	function RUNEXPANDER $(L, k, \alpha, d)$	$l, \text{seed}) \qquad \triangleright L = [l_0, \dots, l_{k-1}] \text{ left nodes}$					
		$\triangleright d$ is degree of right-side nodes					
2:	Initialize PRNG with seed						
3:	$q \leftarrow \alpha k$	$\triangleright$ Number of right-side nodes					
4:	for $0 \le t < q$ do	$\triangleright$ Iterate over all right-side nodes $r_t$					
5:	$acc \leftarrow 0$						
6:	for $0 \le j < d$ do	$\triangleright$ Find d many random left-side nodes $l_i$					
7:	$i \leftarrow \texttt{random}() \mod k$	$\triangleright$ Random node index for L					
8:	$\omega_{i,t} \gets \texttt{field} :: \texttt{random}()$	$\triangleright$ Random field element					
9:	$acc \leftarrow acc + l_i \cdot \omega_{i,t}$	$\triangleright$ Field multiplication with the value of left node					
10:	j++						
11:	$r_t \leftarrow acc$						
12:	<i>t</i> ++						
13:	$\mathbf{return} \ R = [r_0, \dots, r_{q-1}]$						

and causes substantial memory overheads. The memory overhead for a  $N = 2^{28}$  polynomial, which needs 16 iterations of expansion and hence 16 different graphs, is around 1.1 GiB. This high memory demand is a limitation for most hardware systems whereas each graph is only used a single time during encoding.

We address this limitation by combining our inverted expander graphs explained in Section 4.2 with an on-the-fly graph generation. Our on-the-fly generation uses a Pseudo-Random Number Generator (PRNG) to expand all needed graphs based on a small seed. This allows dynamically constructing the inverted expander graphs during the encoding process, as shown in Algorithm 1. Hence, the on-the-fly graph generation seamlessly complies with our inverted expander graphs and eliminates the need for large memory allocations and costly memory accesses. Considering the initial example with  $N = 2^{28}$ , our on-the-fly generation reduces the memory consumption significantly from 1.1GiB to just 8 bytes. In addition, replacing expander graphs is much cheaper using on-the-fly generation as just the 8-byte seed must be exchanged.

## 4.3 Hardware Architecture of Linear Encoding Unit (LEU)

We implement our optimized linear encoding unit for a hardware platform with HBM as the off-chip memory. The architecture is primarily designed for the Xilinx Alveo U280 FPGA, but can also be deployed on similar HBM-coupled FPGAs or ASICs. An overview of our design is presented in Figure 8. The left side in Figure 8 shows the U280 FPGA with its two HBM stacks, whereas each stack provides 16 pseudo-channels. The 32 pseudo-channels are connected via DMA controllers to a total of 32 linear encoding units (LEU).

The overall linear encoding takes the coefficient matrix  $\mathbf{W}$  as input and computes the code matrix  $\mathbf{C}$  in a row-wise manner. Since 32 pseudo-channels are available and  $\mathbf{W}$  has k = 128 rows, we store four rows of  $\mathbf{W}$  in one HBM pseudo-channel (PC) and



Figure 8: Overall Architecture to perform linear encoding in parallel on HBM-based FPGA

dedicate one linear encoding unit (LEU) to each PC. Each LEU operates on one HBM pseudo-channel (PC) and performs the inverted expander graph evaluation for the four rows. A detailed view of a single LEU is provided on the right side of Figure 8. Each LEU features two shift registers for input and output buffering. The MAC unit is responsible for computing the right nodes  $r_t$  by multiplying the randomly read coefficients  $\mathbf{W}[i, j] = w_{i,j}$  (corresponding to the left nodes of the graphs) with the random weights  $\omega_{i,t}$ . Note, that only the first iteration of graph evaluation  $G_1^{(1)}$  takes all the coefficients from a row of  $\mathbf{W}$  as input. Thereafter, the output of the previous graph evaluations is used iteratively as input. The random weights and the random read addresses required in the computations are generated by the Trivium PRNG [Can06] module shown at the bottom of Figure 8.

In the overall encoding process, data is read concurrently from the 32 HBM channels, enabling parallel processing of independent rows. The HBM memory accesses are performed on 256 bits or 512 bits, depending on the clock frequency of the FPGA. This size is larger than the 128-bit size of our extension field elements, hence causing a mismatch between the memory access size (256 or 512 bits) and the 128-bit size of our finite field elements. To cope with this, we pack e.g. 512/128 = 4 elements into one memory word, as shown in Figure 8. When loading one 512-bit word, the input shift register serializes the four elements, which are then processed sequentially. Finally, the four computation results are again packed into one memory word which is stored to the HBM.

This approach effectively uses the available memory capabilities and balances the resource consumption of the LEU units. The shift registers compensate for the latency of random memory reads which typically take around 3 to 4 cycles. This means that a 512-bit input arrives every 4 cycles, whereas it contains 4 field elements. Hence, the LEU is optimally utilized through our data scheduling approach.

## 5 Accelerating Merkle Tree Generation

The generation of the Merkle Tree is the second critical operation in Orion's commitment phase. This operation takes the encoded matrix **C** from linear encoding, as input and requires two steps. In step 1, the columns  $\mathbf{C}[:i] = [c_{0,i} \dots c_{k-1,i}]^T$  are hashed to leaf nodes  $h_{l,i}$ , where l = 0 indicates the leaf level within the Merkle Tree. This step is shown on the left side in Figure 9 on the left. Building upon these leaf nodes, step 2 constructs a Merkle Tree (right in Figure 9). We describe our hardware design for this dual-step computation in the remainder of this section by starting with the column hashing and then advancing to the Merkle Tree computation. Although conceptually simple, implementing a Merkle Tree becomes challenging when handling large datasets stored in off-chip memory.



Figure 9: Computation Flow of Column Hashing and Merkle Tree Generation

### 5.1 Column Hashing

The hashing of columns operates on the linear encoded matrix  $\mathbf{C}$ , which resides in different banks of the HBM memory. The encoded matrix  $\mathbf{C}$  contains k = 128 rows, as specified by the Orion scheme 3.2. According to Section 4, four rows are stored in one HBM pseudo-channel (PCs). This memory layout is shown in Figure 10. In the *i*-th PC, the rows  $\mathbf{C}[4i:]$  to  $\mathbf{C}[4i+3:]$  are stored. Moreover, address *j* in the *i*-th PC holds 4 field elements  $\mathbf{C}[4i:j]$  to  $\mathbf{C}[4i+3:j]$ , which are are packed in a 512-bit memory word. The linear encoding process efficiently utilizes this memory layout as each LEU operates on just one PC. Yet, the column hashing is not directly supported since computing one column hash requires synchronized reading across all PCs to obtain the data. Moreover, a pipelined design is required to optimally exploit the available HBM bandwidth.

To serve the high HBM bandwidth, we instantiate multiple SHA3-256 permutation units (SHA3 Perm), which form our Hash Vector unit shown in Figure 10. However, running multiple SHA3 Perm units in parallel introduces two challenges: (1) efficiently synchronizing the HBM PCs and rearranging the columns to allow for parallel hashing of the leaf nodes, and (2) parallelizing the hashing of multiple columns simultaneously to optimally utilize the high throughput of HBM. We address these two challenges in the next subsections.

### 5.1.1 Pipelined Feeding of Hash Unit

A straightforward possibility to feed the data correctly to the hash unit is to transpose the matrix  $\mathbf{C}$  after encoding and before hashing, thereby placing each column's data into a single PC. This would simplify the memory synchronization for parallel processing, as the data of a single column is placed into one PC only. However, while this approach simplifies memory synchronization, it introduces an additional latency and area overhead caused by the transposing operation.

We instead propose a more suitable approach, which involves partial on-chip buffering and a streamlined data rearrangement of the encoded matrix **C**. We use an on-chip scratchpad buffer that dynamically maps the  $32 \times 512$ -bit read interface to a variable number of 1088-bit SHA3 inputs. Figure 10 shows our Scratchpad Module and its integration into the hashing unit. Through the on-chip buffering, we establish a synchronized reading operation over the 32 PCs. In addition, our method allows for linear and pipelined reading from HBM, providing stall-free input for multiple parallel hashing units.

#### 5.1.2 Scalable Multi-Pass Column Hashing

The scratchpad unit described in the previous section enables the parallelization of column hashing by reshaping the  $32 \times 512$ -bit HBM output to a variable number of 1088-bit



Figure 10: Overview of Computation Flow in the Hash Vector Unit

SHA3-256 inputs. Now that the data is in the right format and ready for absorption into the SHA3-256 state, we can instantiate multiple fully unrolled and pipelined SHA3-256 permutation units, as shown in the bottom right corner of Figure 10. Each permutation unit performs one SHA3 permutation operation together with the data absorption at a rate of 1088 bits. For the column hashing, we need to hash 128 field elements  $c_{i,j}$  each having 128 bits. Therefore, we need to evaluate  $\lceil 128 \cdot 128/1088 \rceil = 16$  SHA3 permutations.

A simple approach for hashing the 16 absorption stages is to instantiate 16 unrolled SHA3 permutation units. This allows the highest throughput but also introduces a substantial area consumption. This high area consumption results from the large, fully unrolled SHA3 Perm units. Moreover, the high area consumption does not justify the performance gain since hashing in Orion accounts for clearly less runtime compared to linear encoding. Alternatively, we offer customized tradeoffs whereby just a power of two SHA3 Perm units is instantiated<sup>1</sup>. For example, Figure 10 shows an architecture with just four SHA3 Perm units. To still hash a full column of **C**, multiple passes through the pipeline are required. The according feedback datapath is also shown in Figure 10. In our experiments, a Hash Vector with up to 8 SHA3 Perm units can be instantiated on the U280 FPGA. Using this configuration, the Hash Vector unit consumes 467k LUTs which is significantly larger than the linear encoding units with 98k LUTs. Yet, this configuration allows for the best performance on the target FPGA.

### 5.2 Merkle Tree Generation

After hashing the columns into leaf nodes  $h_{0,i}$ , the Merkle Tree is constructed. Building the tree involves iteratively hashing pairs of nodes to form parent nodes, as shown in Figure 9. In contrast to the leaf node hashing, the pairs of hashes only have 512 bits and hence can be processed by a single SHA3 permutation unit with its 1088-bit absorption. In addition, at most one leaf node per clock cycle is provided by our pipelined hash vector unit as described above. Based on these observations, we propose a fully pipelined multi-pass architecture called the Merkle Tree Unit (MTU). An overview of the MTU is shown on the right side of Figure 11.

The MTU has one input for the leaf node hashes  $h_{0,i}$  from the Hash Vector unit and one output for the hashes of the Merkle Tree nodes  $h_{l,j}$ , where l is the level of the hash and j the index within the level. The unit operates as follows: it receives a stream of leaf node hashes (256-bit hashes) from the HVU unit. Thereby, just one hash arrives in each clock cycle which requires a single-cycle buffering to aggregate the pair of hashes. After both input hashes are available, they are passed to a fully unrolled, pipelined SHA3-256

<sup>&</sup>lt;sup>1</sup>We support 1, 2, 4, 8, or 16 SHA3 Perm units



Figure 11: Timeline of Multi-Pass-Pipeline during Merkle Tree Generation

permutation unit for hashing. Due to this, the SHA3 Perm pipeline is utilized every second cycle for processing leaf node hashes. This is shown in the top timing diagram in Figure 11. To avoid the shown pipeline bubbles and ensure continuous operation, our architecture issues the level 1 hashing within the bubbles of the leaf node hashing, as soon as level 1 nodes  $(h_{1,j})$  arrive in the feedback path shown in Figure 11. Similar to the leaf node hashing, two hashes are combined in level 1. Therefore, one level 1 hashing operation is issued every four cycles. This is shown in the second timing diagram in Figure 11. We adequately apply this strategy also for the remaining Merkle Tree levels which allows us to fully fill the pipeline of the single SHA3 Perm unit. Thus, the instantiated hardware units are optimally utilized whereby just a negligible control and buffering overhead is introduced.

## 6 Evaluation and Results

In this section, we present the resource utilization and performance results of our Orion commitment accelerator. To collect actual real-world benchmarks, we combine our linear encoding unit from Section 4 and our Hashing unit from Section 5 into an Alveo U280 FPGA with 3 super logic regions (SLRs) and HBM memory [Xil19].

An illustrative overview of our architecture for this FPGA is shown in Figure 12. In terms of placement, the Linear Encoding Wrapper, the DMA controller, and the Scratchpad unit are placed in SLR0. The Hash Vector unit with a total of 8 SHA3 Perm units is spread equally over SLR1 and SLR2. In addition, the SLR2 also instantiates the Merkle Tree unit. The data and control signals are distributed over the FPGA using SLR-crossing indicated by the SLR-crossbar symbolized in Figure 12.

Based on our unified hardware architecture, we present the area and timing results in the remainder of this section. In addition, we compare our results with the software baseline and give benchmarks for an Deep Neural Network (DNN) [APKP24] application that uses Orion for its proof of training.



Figure 12: Overall Architecture on HBM-based FPGA with 3 SLRs

## 6.1 Resource Utilization Results

Table 4 provides a detailed breakdown of the resource utilization for the key hardware components in our implementation. The whole processor design consumes 799k lookup tables (LUTs), 570k registers (REGs), 1.2k digital signal processors (DSPs), and 334 block RAMs. No Ultra RAMs are used in our design. The overall Processor consists of Platform logic which mainly establishes the HBM and PCI connection and the actual Cryptoprocessor containing our Orion accelerator.

Within our Cryptoprocessor, which causes the major area consumption, the remaining modules of DMA, linear encoding, and hashing are instantiated. As described previously, the DMA, the Scratchpad, and the linear encoding reside in the first SLR. This causes a resource utilization in SLR0 of 39%, 21%, 38%, and 38% in LUTs, REGs, DSPs, and BRAMs, respectively. The SLR1 holds half of the Hash Vector Unit and has a utilization of 53% and 13% of LUTs and REGs, respectively. Finally, SLR2 has the remaining half of the Hash Vector Unit and the Merkle Tree unit. This accounts for 68% and 16% LUT and REG utilization in SLR2.

Although our Scratchpad unit architecture can support any power of two number of SHA3 Perm units within the Hash Vector unit, we can only fit at most 8 of them into the U280 FPGA due to the large size of the hashing subsystem. In addition to that, the hashing and Merkle Tree generation accounts for 80% of the LUT consumption of the Cryptoprocessor whereas the Linear Encoding Wrapper only causes 14% of the LUT consumption.

These numbers show the different limitations of these operations; the hashing is resourcebound and therefore requires more hardware resources to maintain high performance. Compared to the hashing, the runtime of the linear encoding is memory-bound due to the high number of random HBM accesses. Thus, dedicating more hardware resources to linear encoding does not contribute to performance. Instead, ingenious solutions to efficiently use the memory bandwidth are required. This highlights the relevance of our inverted expander graphs which enhance the effective memory bandwidth.

## 6.2 Timing Results

Table 5 outlines the performance results of our hardware-accelerated linear encoding and hashing within the Orion protocol. The results were collected for the Alveo U280 FPGA running on 225MHz. The provided results cover a range of input polynomial sizes from

Modules	#Units	LUTs	REGs	DSPs	BRAMs
Alveo U280 FPGA Total	3 SLRs	1,304k	2,607k	9k	2k
Processor	1	799,391	$569,\!849$	$1,\!156$	334
- Platform	1	$101,\!639$	$131,\!244$	4	70
- Cryptoprocessor	1	697,752	$438,\!605$	$1,\!152$	264
DMA Controller	1	40,768	93,792	-	-
- Read Interface	32	704	1,502	-	-
- Write Interface	32	570	$1,\!429$	-	-
Linear Encoding Wrapper	1	98,272	63,360	$1,\!152$	-
- Linear Encoding Unit	32	3,071	$1,\!980$	36	-
Scratchpad	1	30,778	25,856	-	256
Hash Vector Unit	1	466,977	$228,\!025$	-	-
- SHA3 Perm Unit	8	55,361	23,752	-	-
Merkle Tree Unit	1	60,957	$27,\!572$	-	8
- SHA3 Perm Unit	1	$55,\!361$	23,752	-	-

Table 4: Resource Utilization of Modules on Alveo U280 FPGA

 $N = 2^{16}$  to  $N = 2^{28}$ . For a given polynomial size N, the encoding result is approximately twice the size, as reported in Table 5.

The linear encoding latency in our architecture for  $N = 2^{16}$  is 177 microseconds and reaches up to 730 milliseconds for  $N = 2^{28}$ . When N is doubled, the linear encoding latency also roughly doubles, which is expected in the Orion scheme. Considering the hashing and Merkle Tree generation latency, we provide different benchmarks for the supported number of SHA3 Perm units in the Hash Vector in Table 5. The slowest and most lightweight configuration with 1 SHA3 Perm has a latency between 73 microseconds and 298 milliseconds while the largest configuration has a latency between 5 microseconds and 18 milliseconds for N between  $2^{16}$  and  $2^{28}$ . The hashing latency is also linear in the polynomial size – similar to linear encoding – due to our optimized Merkle Tree generation with interleaved node scheduling. In addition, increasing the number of SHA3 Perm units leads to a linear decrease in hashing latency thereby improving performance through higher resource consumption.

Table 5 clearly shows that the linear encoding takes approximately between  $2.4 \times$  and  $40 \times$  longer than hashing, depending on polynomial size and number of SHA3 Perm units. Yet, while the latency of hashing can be reduced by using more SHA3 Perm units, the latency of linear encoding is memory-bound. This makes the further acceleration of linear encoding highly challenging given the constraints of memory layout and operation flow in the Orion scheme. It is noteworthy, that hashing accounts for a fraction of the total runtime while consuming most of the resources meaning that an adequate number has to be chosen depending on the use case.

## 6.3 Comparison with Baseline of Orion

In this section, we compare the performance of our hardware implemented on an Alveo U280 FPGA with the original software implementation [Ori] from [XZS22]. The comparison focuses on the critical linear encoding and the Merkle Tree generation. In addition, we report the overall speedup for Orion's commitment and give an application benchmark for DNN training. Note that no hardware accelerator has been published targeting the Orion ZKP scheme or proposing inverted expander graphs. Hence, we focus on the software implementation of Orion to show our achieved improvements. The software results were collected by running Orion on an AMD EPYC 9754 @2.25GHz server CPU with 128 cores. Due to the high core count of this server CPU supports a much lower base clock frequency and max boost clock frequency (Turboboost) technology unlike the Intel server CPU (Intel

T t G!	<b></b>	<b>T T U</b>	TT 1	• • •		<b>T</b> (*	<u> </u>
Input Size	Encoded	Lin. Encoding	Hash	ling and 1	Vierkle	Tree (in	ms)
$\log(N)$	Size	(in ms)	$1^\dagger$	$2^{\dagger}$	$4^{\dagger}$	$8^{\dagger}$	$16^{\dagger *}$
16	17	0.177	0.073	0.036	0.018	0.009	0.005
17	18	0.355	0.146	0.073	0.036	0.018	0.009
18	19	0.712	0.291	0.146	0.073	0.036	0.018
19	20	1.424	0.582	0.291	0.146	0.073	0.036
20	21	2.851	1.164	0.582	0.291	0.145	0.073
21	22	5.703	2.330	1.165	0.582	0.291	0.146
22	23	11.408	4.660	2.330	1.165	0.582	0.291
23	24	22.818	9.320	4.660	2.330	1.165	0.582
24	25	45.637	18.639	9.320	4.660	2.330	1.165
25	26	91.276	37.279	18.639	9.320	4.660	2.330
26	27	182.554	74.558	37.279	18.639	9.320	4.660
27	28	365.109	149.000	74.000	37.000	18.000	9.000
28	29	730.220	298.000	149.000	74.000	37.000	18.000

Table 5: Latency of our FPGA Prototype for Linear Encoding and Hashing

 $\dagger$  Number of SHA3 Perm units within the Hash Vector; \*Results for 16 SHA3 Perm units are extrapolated and not implemented on U280 due to the high resource utilization.

Xeon Platinum 8375C with 32 cores) used by the authors of Orion that runs on 2.90GHz. Hence, the timing we report for the software baseline is slightly different compared to the Orion paper.

### 6.3.1 Linear Encoding Comparison

Table 6 compares the linear encoding timing of our hardware accelerator with the reference software implementation for various polynomial sizes N. Since the linear encoding in our hardware design does not use the hashing unit, the results discussed in Table 6 are independent of the number of instantiated SHA3 Perm units.

As Table 6 shows, we achieve a speedup of  $142 \times \text{compared to software-based encoding}$ for small  $N = 2^{16}$ -degree polynomials. As the polynomial size N increases, our speedup reaches  $381 \times$  for N = 28. Hence, we observe that the speedup grows with the polynomial size N. This is explained by the linear dependency between N and encoding latency in our hardware design whereas the software shows a super-linear dependency of latency on N. This super-linear dependency of software originates from the increasingly costly random memory accesses where the caching in the CPU becomes less effective for larger N. In contrast to that, our timing shows a linear increase with N. We achieve this by not relying on caching and using our novel graph inversion techniques. This allows us to reduce the costly random memory reads and writes which effectively improves the linear encoding performance. In addition to the graph inversion, we leverage the performance of linear encoding by using HBM memory with efficient memory management and on-the-fly graph generation. The on-the-fly graph generation its data via a PRNG and does not involve any memory access therefore reducing the pressure on the memory even further compared to the original implementation.

#### 6.3.2 Merkle Tree Generation Comparison

In Table 7, we present the latency comparison of hashing and Merkle Tree generation between the software baseline and our optimized hardware design. Our flexible hardware configurations of the leaf node hashing lead to different speedups in the Merkle Tree generation. The different speedups for 1, 2, 4, and 8 SHA3 Perm units are experimentally determined while the configuration with 16 SHA3 Perm units was extrapolated as it is too large for the Alveo U280 FPGA.

Input Size	Encoded	Ref. SW	Our HW	Our
$\log(N)$	$\mathbf{Size}$	(in ms)	(in ms)	$\mathbf{Speedup}$
16	17	25	0.177	142
17	18	52	0.355	145
18	19	104	0.712	147
19	20	212	1.424	149
20	21	463	2.851	162
21	22	1,052	5.703	184
22	23	2,234	11.408	196
23	24	4,778	22.818	209
24	25	10,080	45.637	221
25	26	21,323	91.276	234
26	27	45,855	182.554	251
27	28	100,400	365.109	275
28	29	$278,\!100$	730.220	381

Table 6: Performance Comparison for Linear Encoding: Our Design on Alveo U280 @225MHz (HW) vs Reference Software on AMD EPYC 9754 @2.25GHz (SW).

When configured with a single SHA3 Perm unit, our hardware provides a 144× speedup for larger polynomial sizes, such as  $N = 2^{28}$ , compared to the software baseline. The time taken for Hashing and Merkle Tree generation in this minimal configuration is 298 milliseconds (as reported in Table 5) while the software implementation requires 43.016 seconds. Furthermore, the performance improves significantly as we scale up the number of SHA3 Perm units. In the case of 8 units, which is the maximum on the target FPGA, the latency for Hashing and Merkle Tree generation is even further reduced. This results in a significant speedup over the software between  $1,122 \times$  and  $1,189 \times$  for various polynomial degrees.

Note that the scalability of our architecture is one of its key strengths. It allows us to optimize the system for different hardware environments by adjusting the number of hash units. Thus, we can support smaller FPGAs with limited resources but also larger high-performance platforms such as the U280 FPGA. In contrast, the software baseline is limited by the inherent architecture of the underlying processor. Overall, the flexibility and scalability of our hardware design ensure that both Hashing and Merkle Tree generation can be tailored to meet the performance requirements of any system, while still outperforming the software in all configurations.

### 6.3.3 Overall Commitment Comparison

The overall commitment process in the Orion protocol includes both linear encoding and Merkle Tree generation. Hence, the overall latency of commitment in our hardware design is the latency of linear encoding plus the latency of hashing and Merkle Tree generation. Table 8 compares the timing results of our hardware implementation and those of the software baseline [Ori].

Compared to the software-based commitment, our hardware design shows a significant speedup across all tested polynomial sizes. For smaller polynomial sizes such as  $N = 2^{16}$ , our hardware with 8 SHA3 Perm units has a total commitment time of 0.19 milliseconds, whereas the software needs 40 milliseconds. This results in a  $216 \times$  speedup for small polynomials. As the polynomial size increases, the performance gap widens further. For  $N = 2^{28}$ , our hardware accelerator completes the entire commitment process in just 767.22 milliseconds, compared to 334.25 seconds in the software. Thus, we achieve an overall speedup of  $436 \times$ . The key factor behind this significant improvement is the combination of our proposed optimizations in both linear encoding, leaf node hashing, and Merkle Tree

Input Size	Encoded	Ref. SW		Our S	Speed	up in H	IW
$\log(N)$	Size	(in ms)	$1^\dagger$	$2^{\dagger}$	$4^{\dagger}$	$8^{\dagger}$	$16^{\dagger *}$
16	17	10	138	281	561	1,122	2,020
17	18	21	144	288	577	$1,\!154$	$2,\!307$
18	19	41	141	282	563	$1,\!126$	$2,\!252$
19	20	82	141	282	563	$1,\!126$	$2,\!252$
20	21	165	142	284	568	$1,\!139$	2,263
21	22	333	143	286	572	$1,\!143$	2,287
22	23	664	142	285	570	$1,\!140$	$2,\!280$
23	24	$1,\!330$	143	285	571	$1,\!142$	$2,\!283$
24	25	$2,\!679$	144	287	575	$1,\!150$	$2,\!300$
25	26	$5,\!352$	144	287	574	$1,\!149$	$2,\!297$
26	27	10,709	144	287	575	$1,\!149$	2,298
27	28	$21,\!397$	144	289	578	$1,\!189$	$2,\!377$
28	29	43,014	144	289	581	1,163	2,390

Table 7: Performance Comparison for Hashing and Merkle Tree: Our Design on AlveoU280 @225MHz (HW) vs Reference Software [Ori] on AMD EPYC 9754 @2.25GHz (SW).

 $\dagger$  Number of SHA3 Perm units within the Hash Vector; \*Results for 16 SHA3 Perm

units are extrapolated and not implemented on U280 due to high resource utilization.

Table 8: Performance Comparison for whole Commitment: Our Design on Alveo U280 @225MHz (HW) vs Reference Software on AMD EPYC 9754 @2.25GHz (SW).

Size	Ref. SW	(	Our Latency	(in ms) / Spe	edup in HW	
$\log(N)$	(in ms)	$1^{\dagger}$	$2^{\dagger}$	$4^{\dagger}$	$8^{\dagger}$	$16^{\dagger *}$
16	40	0.25 / 161	0.21 / 189	0.20 / 206	0.19 / 216	0.18 / 221
18	163	1.00 / 163	0.86 / 190	0.79 / 208	0.75 / 218	0.73 / 224
20	701	4.02 / 175	3.43 / 204	3.14 / 223	3.00 / 234	2.92 / 240
22	3175	16.07 / 198	13.74 / 231	12.57 / 253	11.99 / 265	11.70 / 271
24	13,756	64.28 / 214	54.96 / 250	50.30 / 273	47.97 / 287	46.80 / 294
26	60,011	257.11 / 233	219.83 / 273	201.19 / 298	191.87 / 313	187.21 / 321
28	$334,\!250$	1,028.22 / 325	879.22 / 380	804.22 / 416	767.22 / 436	748.22 / 447

<sup>†</sup> Number of SHA3 Perm units within the Hash Vector; \*Results for 16 SHA3 Perm units are extrapolated and not implemented on U280 due to high resource utilization.

generation. In contrast, the software baseline faces limitations due to its sequential nature and less efficient memory management. This especially affects larger polynomial sizes, resulting in much longer commitment times in the software case.

Overall, we reach more than two orders of magnitude lower latencies for Orion's commitment across different parameter sets. These results show that our hardware acceleration architecture is not only faster across individual components but also leads to a significant reduction in the overall commitment time. This high-speed speedup is crucial for applications requiring large zero-knowledge proofs, wherein reducing the total commitment time directly translates into improved system performance and scalability.

## 6.4 Accelerating Training of Deep Neural Networks

Training deep neural networks (DNNs) with privacy-preserving methods, such as zeroknowledge proofs of training (zkPoTs), involves verifying the correctness of the model's training without revealing sensitive information. One of the key factors influencing performance in these systems is the batch size B, which refers to how many training samples are processed together in a single iteration. Larger batch sizes increase computational demands, as more data must be committed and verified during each iteration. The work in [APKP24] shows that the commitment phase in zkPoTs becomes increasingly demanding

Proofs for		LeNet		;		
different DNNs	B=4	B = 8	B = 16	B=4	B = 8	B = 16
Commitment in SW	70.04	93.46	106.6	94.72	139.7	233.7
Total Prover (sec)	112.4	141.8	193.4	262.9	332.6	474.4
Commitment in HW	0.050	0.066	0.134	0.268	0.403	0.673
Total Prover (sec)	42.1	48.4	86.9	168.5	193.3	241.4
Speedup Commit	1408  imes	1416  imes	796  imes	353 imes	346  imes	347
Speedup Proof	2.673 imes	$\pmb{2.929}  imes$	2.224 imes	1.561×	1.720	2.029 imes

Table 9: Application Benchmarks for zkPoTs of Deep Neural Networks [APKP24, Kai]

as the batch size grows, particularly for applications like LeNet and AlexNet.

The commitment phase is a critical step in zkPoTs, where model parameters and gradients are committed securely during each training iteration. More parameters need to be committed as the batch size increases, thereby increasing the computational burden. Our hardware accelerator addresses this challenge by introducing a novel graph generation technique, which drastically improves the efficiency of the commitment process. The results in Table 9 demonstrate that our architecture achieves a  $347 \times$  to  $1,416 \times$  speedup in the commitment phase compared to the software baseline. This improvement is largely due to our new graph generation method, which replaces the conventional approach with a highly efficient, hardware-optimized algorithm. The traditional method of generating graphs for commitment involves multiple recursive steps that introduce significant overhead. Our approach eliminates these bottlenecks by generating the graph structure in an iterative, hardware-friendly manner, allowing it to be processed in parallel across multiple units.

While the commitment phase benefits the most from our optimizations, the overall zkPoT proof generation process also experiences improvements. As shown in Table 7, the total speedup for the full proof generation process ranges from  $1.561 \times$  to  $2.929 \times$ , depending on the batch size and model. Although this improvement is more moderate compared to the commitment phase alone, it still represents an important enhancement in overall zkPoT efficiency, particularly for larger models and higher batch sizes.

Even modest gains in overall proof generation can have a notable impact when scaling to real-world applications. In privacy-preserving machine learning tasks like federated learning or secure multi-party computation, where multiple training iterations occur, these speedups reduce the overall time to verify model updates, making zkPoTs more practical for large-scale training scenarios.

In summary, our hardware accelerator handles the increased computational demands of larger batch sizes in zkPoTs efficiently. The  $347 \times to 1,416 \times$  speedup in the commitment phase eliminates a key bottleneck, while the overall proof generation process benefits from a  $1.561 \times to 2.929 \times$  speedup. These improvements make zkPoT-based deep learning systems faster and more scalable, ensuring they can handle large-scale training tasks without compromising on speed or privacy.

## 7 Conclusion

This work introduced several optimization techniques aimed at improving both the algorithmic efficiency and the hardware implementation of Orion. As demonstrated in Section 3.3, the commitment phase is the most computationally expensive part of the Orion proof system. Enhancing the efficiency of this phase is critical to improving Orion's overall performance, and our optimizations specifically target this bottleneck. We addressed the high runtime by optimizing the sub-operations within the commitment phase, namely graph-based linear encoding and Merkle Tree generation.

For linear encoding, we developed a novel iterative approach, replacing the recursive

graph expansion algorithm with one better suited to hardware implementation. By inverting the graph generation process, we significantly reduced memory consumption from several gigabytes to just a few 64-bit seeds resulting in substantial power and latency savings due to reduced off-chip memory accesses. Our linear encoding results show speedups ranging from  $142 \times$  to  $381 \times$ , depending on the size of the input polynomial. For Merkle Tree generation, we developed a scalable methodology for hashing and tree generation. This scalable methodology allows for more compatibility across both high-performance systems and resource-constrained platforms. The scalability in the hashing allows for a section of 1 to 16 SHA3 units. Depending on the number of units, we achieved speedups ranging from  $138 \times$  to  $2,390 \times$  compared to the reference design.

The architectural optimizations introduced in this work led to substantial improvements in Orion's commitment phase, particularly by reducing the computational latency. Our FPGA implementation served as a proof of concept, demonstrating the practicality and efficiency of these enhancements in hardware. The results clearly illustrate the potential of this optimized architecture to accelerate prover-side operations, making Orion more suitable for real-world applications.

## 8 Future Work

There are several avenues for future research building on the findings of this work. First, adapting these optimizations for ASIC implementations could further enhance performance and energy efficiency, particularly for large-scale ZKP deployments. Additionally, optimizing the loading pattern in linear encoding to take advantage of larger AXI bursts, instead of relying on single random accesses, could yield further performance gains by better utilizing memory bandwidth. Extending these optimizations to other zero-knowledge proof systems beyond Orion could unlock new opportunities to improve the efficiency of cryptographic protocols in privacy-preserving technologies. Finally, exploring the potential for implementing full proof generation entirely on hardware would be a promising direction. However, the heterogeneous nature of the computations involved in proof generation – involving various arithmetic, logic, and memory access patterns – presents a significant challenge in achieving this goal.

## References

- [APKP24] Kasra Abbaszadeh, Christodoulos Pappas, Jonathan Katz, and Dimitrios Papadopoulos. Zero-knowledge proofs of training for deep neural networks. Cryptology ePrint Archive, Paper 2024/162, 2024.
- [APPK24] Kasra Abbaszadeh, Christodoulos Pappas, Dimitrios Papadopoulos, and Jonathan Katz. Zero-knowledge proofs of training for deep neural networks. Accepted in ACM CCS 2024, page 162, 2024.
- [BBB<sup>+</sup>18] Benedikt Bünz, Jonathan Bootle, Dan Boneh, Andrew Poelstra, Pieter Wuille, and Gregory Maxwell. Bulletproofs: Short Proofs for Confidential Transactions and More. In 2018 IEEE Symposium on Security and Privacy, SP 2018, Proceedings, 21-23 May 2018, San Francisco, California, USA, pages 315–334. IEEE Computer Society, 2018.
- [BCG20] Jonathan Bootle, Alessandro Chiesa, and Jens Groth. Linear-time arguments with sublinear verification from tensor codes. In Theory of Cryptography: 18th International Conference, TCC 2020, Durham, NC, USA, November 16–19, 2020, Proceedings, Part II 18, pages 19–46. Springer, 2020.

- [Big02] Norman Biggs. Discrete Mathematics. OUP Oxford, 12 2002.
- [BSCG<sup>+</sup>14] Eli Ben Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized Anonymous Payments from Bitcoin. In 2014 IEEE Symposium on Security and Privacy, pages 459–474, 2014.
- [Can06] Christophe De Cannière. Trivium: A stream cipher construction inspired by block cipher design principles. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 4176 LNCS:171–186, 2006.
- [CHM<sup>+</sup>20] Alessandro Chiesa, Yuncong Hu, Mary Maller, Pratyush Mishra, Psi Vesely, and Nicholas P. Ward. Marlin: Preprocessing zksnarks with universal and updatable SRS. In Anne Canteaut and Yuval Ishai, editors, Advances in Cryptology - EUROCRYPT 2020 - 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10-14, 2020, Proceedings, Part I, volume 12105 of Lecture Notes in Computer Science, pages 738–768. Springer, 2020.
- [CKLE20] Benjamin Y. Cho, Yongkee Kwon, Sangkug Lym, and Mattan Erez. Near data acceleration with concurrent host access. In 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pages 818–831, 2020.
- [Fou] Zcash Foundation. Zcash digital currency. Accessed on October 2024.
- [FS86] Amos Fiat and Adi Shamir. How to Prove Yourself: Practical Solutions to Identification and Signature Problems. In Andrew M. Odlyzko, editor, Advances in Cryptology - CRYPTO 1986, volume 263 of LNCS, pages 186–194. Springer, 1986.
- [GLS<sup>+</sup>21] Alexander Golovnev, Jonathan Lee, Srinath Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and field-agnostic snarks for r1cs. Cryptology ePrint Archive, 2021.
- [GLS<sup>+</sup>23] Alexander Golovnev, Jonathan Lee, Srinath T. V. Setty, Justin Thaler, and Riad S. Wahby. Brakedown: Linear-time and field-agnostic snarks for R1CS. In Helena Handschuh and Anna Lysyanskaya, editors, Advances in Cryptology -CRYPTO 2023 - 43rd Annual International Cryptology Conference, CRYPTO 2023, Santa Barbara, CA, USA, August 20-24, 2023, Proceedings, Part II, volume 14082 of Lecture Notes in Computer Science, pages 193–226. Springer, 2023.
- [GMR85] Shafi Goldwasser, Silvio Micali, and Charles Rackoff. The knowledge complexity of interactive proof-systems (extended abstract). In Robert Sedgewick, editor, Proceedings of the 17th Annual ACM Symposium on Theory of Computing, May 6-8, 1985, Providence, Rhode Island, USA, pages 291–304. ACM, 1985.
- [Gol01] Oded Goldreich. Foundations of cryptography. *Foundations of Cryptography*, 8 2001.
- [Gro16] Jens Groth. On the Size of Pairing-based Non-interactive Arguments. *IACR Cryptol. ePrint Arch.*, page 260, 2016.

- [GY18] Hisham S. Galal and Amr M. Youssef. Verifiable sealed-bid auction on the ethereum blockchain. In *Financial Cryptography and Data Security - FC 2018*, volume 10958 of *LNCS*, pages 265–278. Springer, 2018.
- [Hei03] James L. Hein. *Discrete Mathematics*. Jones and Bartlett Learning, 2003.
- [HP10] W. Cary Huffman and Vera Pless. *Fundamentals of Error-Correcting Codes*. Cambridge University Press, 2 2010.
- [Hub19] Xilinx Developer Hub. Basic tutorial for maximizing memory bandwidth with vitis and xilinx ultrascale+ hbm devices, 11 2019.
- [Kai] zkpots/kaizen. https://github.com/zkPoTs/kaizen/tree/main. Accessed: 2024-10-12.
- [KZG10] Aniket Kate, Gregory M. Zaverucha, and Ian Goldberg. Constant-size commitments to polynomials and their applications. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 6477 LNCS:177–194, 2010.
- [Lee21] Jonathan Lee. Dory: Efficient, Transparent Arguments for Generalised Inner Products and Polynomial Commitments. In Kobbi Nissim and Brent Waters, editors, *Theory of Cryptography*, pages 1–34, Cham, 2021. Springer International Publishing.
- [LWX<sup>+</sup>23] Xiling Li, Chenkai Weng, Yongxin Xu, Xiao Wang, and Jennie Rogers. ZKSQL: verifiable and efficient query evaluation with zero-knowledge proofs. *Proc. VLDB Endow.*, 16(8):1804–1816, 2023.
- [LXZ21] Tianyi Liu, Xiang Xie, and Yupeng Zhang. zkCNN: Zero Knowledge Proofs for Convolutional Neural Network Predictions and Accuracy. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021, pages 2968–2985. ACM, 2021.
- [Ori] sunblaze-ucb/orion. https://github.com/sunblaze-ucb/Orion. Accessed: 2024-09-30.
- [Pro] The Monero Project. Monero secure, private, untraceable. https://www.getmonero.org/, Accessed on October 2024.
- [RL09] William Ryan and Shu Lin. Channel Codes: Classical and Modern. Cambridge University Press, 9 2009.
- [RZR24] Noga Ron-Zewi and Ron Rothblum. Local proofs approaching the witness length. J. ACM, 71(3), June 2024.
- [Spi96] Daniel A. Spielman. Linear-time encodable and decodable error-correcting codes. *IEEE Transactions on Information Theory*, 42:1723–1731, 1996.
- [SS94] M. Sipser and D.A. Spielman. Expander codes. In Proceedings 35th Annual Symposium on Foundations of Computer Science, pages 566–576. Institute of Electrical and Electronics Engineers (IEEE), 12 1994.
- [WTS<sup>+</sup>18] R. S. Wahby, I. Tzialla, A. Shelat, J. Thaler, and M. Walfish. Doubly-Efficient zkSNARKs Without Trusted Setup. In 2018 IEEE Symposium on Security and Privacy (SP), pages 926–943, Los Alamitos, CA, USA, may 2018. IEEE Computer Society.

- $[WYX^+21]$ Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient Conversions for Zero-Knowledge Proofs with Applications to Machine Learning. In Michael Bailey and Rachel Greenstadt, editors, 30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021, pages 501–518. USENIX Association, 2021.
- [Xil19] Xilinx. Alveo u280 data center accelerator card data sheet, 2019. https://www.avnet.com/opasdata/d120001/medias/docus/196/ XLX-A-U280-A32G-DEV-G-Datasheet.pdf.
- [Xil24] Xilinx. Hbm system considerations, 2024. https://docs.amd.com/r/en-US/ pg313-network-on-chip/HBM2E-Access-Latencies, accessed on October 2024.
- [XZS22]Tiancheng Xie, Yupeng Zhang, and Dawn Song. Orion: Zero knowledge proof with linear prover time. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 13510 LNCS:299-328, 2022.

#### Α Algorithms & Protocols of Orion

Algorithm 2 Searching Non-expanding Set [XZS22]

- 1: Let G = (V, E) with L and R the left and right vertex sets of a random bipartite graph.  $\triangleright \alpha = \frac{|R|}{|L|}$
- 2: If  $\exists v \in R$  with degree  $d \geq \frac{c}{\alpha} + 10 \ln k$ , abort.
- 3: for each  $v \in L$  do
- 4: find set  $D \subseteq L$  such that: -  $\forall u \in D$  the minimum distance between u and v is  $\leq 2 \log \log k$ -  $\forall u \in L \setminus D$  the minimum distance between u and v is  $> 2 \log \log k$ for all  $S \subseteq D$  and  $|S| \leq \log \log k$  do 5:if Equation 1 does not hold for S then 6: return Found 7: 8: return NotFound

#### Β Baseline Expander Graph Generation

The baseline expander graph G = (L, R, E) as used in Orion, is generated and stored in memory. The generation algorithm is shown in Algorithm 3. It can be seen that the graph is stored as a set E of tuples of the form  $(l_i, r_t, \omega_{i,t})$ . Each tuple describes one edge in the graph staring at left node  $l_i \in L$  and connecting the right node  $r_t \in R$  with a weight  $\omega_{i,t}$ . Unlike in our inverted expander graphs, the degree of left nodes c is constant in Algorithm 3.

Algorithm 3 Baseline Expander Graph Generation Algorithm from [XZS22]

1: function GRAPHGEN $(L, R, c) \triangleright L$ , R set of left & right nodes, c is left nodes degree  $k \leftarrow |L|$ 2:  $\triangleright q = |R| = \alpha |L|$  $q \leftarrow \alpha k$ 3:  $E \leftarrow \{\}$ 4: for  $0 \le i < k$  do 5:for  $0 \le j < c$  do 6: 7:  $t \leftarrow \texttt{rand}() \mod q$  $\triangleright$  A random vertex index in right subset  $\omega_{i,t} \leftarrow \texttt{field} :: \texttt{random}()$ 8:  $\triangleright$  A random field element Add edge  $(l_i, r_t, \omega_{i,t})$  from left node  $l_i$  to right node  $r_t$  with weight  $\omega_{i,t}$  to E 9: j++ 10: 11: return E

## C Statistical Analysis of Expander Graphs

To quantify the properties of an expander graph G = (L, R, E), we use several statistical tools, focusing on the minimum, maximum, mean, and variance of the degree of vertices. Therein, we denote the degree of left vertices  $l \in L$  and right vertices  $r \in R$  as c and d, respectively.

In the first step of the analysis, we consider baseline expander graphs with fixed c and normally distributed d. Orion uses two configurations of expanders,  $G_1^{(i)}$  and  $G_2^{(i)}$ , with  $\{c_1, d_1, \alpha_1\}$  and  $\{c_2, d_2, \alpha_2\}$ , respectively. For both graph types, we experimentally found the statistical properties as in Table 10 for different N. Note, that the values for  $c_1 = 10$ and  $c_2 = 20$  are fixed according to Orion's specifications.

More formally, in graph  $G_1^{(i)}$ , the expected value  $E(d_1)$  of the right-side degree  $d_1$  can be computed as in Equation 4 for some polynomial size N. For graph  $G_2^{(i)}$ , the expected value  $E(d_2)$  is computed as in Equation 5 [XZS22].

$$E(d_1) = \frac{N \cdot c_1}{\lfloor N \cdot \alpha_1 \rfloor} = \frac{N \cdot 10}{\lfloor N \cdot 0.238 \rfloor}$$
(4)

$$E(d_2) = \frac{N' \cdot c_2}{\lfloor N \cdot \alpha_2 \rfloor} = \frac{N' \cdot 20}{\lfloor N \cdot 0.31 \rfloor}$$
(5)

Therein,  $N' = \lfloor \lfloor N \cdot \alpha_1 \rfloor \cdot r \rfloor = \lfloor \lfloor N \cdot 0.238 \rfloor \cdot 1.72 \rfloor$  for the scheme parameter r. The results of the theoretical computation match the mean results of our experiments reported in Table 10. In addition, the expected values  $E(d_1)$  and  $E(d_2)$  only vary negligibly with respect to N.

Building on the statistical properties of the reference graphs, especially on  $E(d_1)$  and  $E(d_2)$ , we find the parameters for Inverted expander graphs. To compute our inverted expander graphs, we set  $d_1$  and  $d_2$  to the rounded expected values, namely  $d_1 = 42$  and  $d_2 = 26$ . With that, our inverted expander graphs are sufficiently parametrized. The resulting normal distribution of  $c_1$  using our inverted expander graph with  $d_1 = 42$  is shown in Figure 6. Similar investigations can be done for the Normal and the Without Replacement methods. By using the mean and standard deviations in Table 10, the parameters for Normal and Without Replacement methods can be easily found.

Table 10: Statistical properties for baseline expander graphs  $G_1^{(i)}$  and  $G_2^{(i)}$ 

Table 10. Statistical properties for backfield expander graphs $O_1$ and $O_2$									u 0 <sub>2</sub>
Ν	$c_1 \operatorname{Min} d_1 \operatorname{Min}$	$c_1 \operatorname{Max} d_1 \operatorname{Max}$	$c_1$ Mean $d_1$ Mean	$c_1$ Variance $d_1$ Variance	N	$\begin{array}{c} c_2 \ \mathrm{Min} \\ d_2 \ \mathrm{Min} \end{array}$	$c_2 \operatorname{Max} d_2 \operatorname{Max}$	$c_2$ Mean $d_2$ Mean	$c_2$ Variance $d_2$ Variance
$2^{15}$	10	10	10.0	0.0	$2^{15}$	20	20	20.0	0.0
-	21	67	42.0	40.9	-	8	47	26.3	26.5
<b>9</b> 15	10	10	10.0	0.0	<b>9</b> 15	20	20	20.0	0.0
2	21	72	42.0	42.3	2	8	47	26.3	27.0
<b>9</b> 15	10	10	10.0	0.0	<b>9</b> 15	20	20	20.0	0.0
2	22	67	42.0	41.5	2	8	51	26.3	26.4
<b>9</b> 20	10	10	10.0	0.0	<b>n</b> 20	20	20	20.0	0.0
2	18	72	42.0	42.0	2	7	51	26.4	26.4
<b>9</b> 20	10	10	10.0	0.0	<b>9</b> 20	20	20	20.0	0.0
2	16	77	42.0	42.0	2	6	57	26.4	26.3
<b>9</b> 20	10	10	10.0	0.0	220	20	20	20.0	0.0
	17	75	42.0	41.7		6	57	26.4	26.3

(a) Graph  $G_1^{(i)}$  with fixed  $c_1 = 10$  (b) Graph  $G_2^{(i)}$  with fixed  $c_2 = 20$