Sluice: Prove-Phase Bounded-Memory Groth16 via Read-Write Streaming

Kyeongtae Lee
Kookmin University
Seoul, Republic of Korea
sklee63kr@gmail.com

Jihye Kim
Kookmin University
Seoul, Republic of Korea
jihyek@kookmin.ac.kr

Hyunok Oh
Hanyang University
Seoul, Republic of Korea
hoh@hanyang.ac.kr

***Abstract*—We present Sluice, a read-write streaming Groth16 prover that reduces *prove-phase* random-access working memory from $\mathcal{O}(N)$ to $\mathcal{O}(\log N)$ once the CRS, QAP, and witness are materialized as private streams. It preserves the standard Groth16 interface: a proof of 3 group elements, 3-pairing verification, and unchanged verifier contracts.**

Our key technical contribution is *Split-Butterfly-Merge* (SBM), an NTT algorithm in the read-write streaming model with $\mathcal{O}(\log N)$ memory, $\mathcal{O}(N \log N)$ total I/O, and $\mathcal{O}(\log N)$ sequential passes over external storage.

Combining SBM with streaming sparse R1CS evaluation and chunked Pippenger MSM yields a verifier-compatible Groth16 proving path that exchanges RAM for sequential storage I/O and wall-clock time. Our prototype uses a fixed-window MSM engineering point; the measurements validate memory reduction and proof compatibility, while the theorem states the asymptotically tuned MSM schedule.

We implement Sluice over BN-254. Direct prove-only runs produce valid 128-byte proofs through $N = 2^{25}$. The same-size bounded-memory comparison is at $N = 2^{23}$: Sluice succeeds under an 8GB Linux cgroup cap, whereas the standard prover is killed under 8GB and 12GB caps and succeeds only at 16GB. These results position Sluice as a storage-rich, RAM-limited proving option rather than a replacement for optimized in-memory provers.

1. Introduction

Zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs) are now used in blockchain scaling systems, verifiable computation, and other settings where succinct verification matters. Among deployed schemes, Groth16 [1] remains especially attractive: its proof consists of only 3 group elements and verification requires only 3 pairings. These constants make Groth16 the dominant choice when on-chain verification cost is the bottleneck.

However, after padding d R1CS constraints to an N-point QAP/NTT domain, the Groth16 prover requires $\mathcal{O}(N)$ random-access memory. For multi-million-constraint circuits, this can exceed commodity RAM budgets, making proving infeasible on RAM-limited but storage-rich prover hosts. The memory bottleneck arises from two sources: the Number Theoretic Transform (NTT), which uses $\mathcal{O}(N)$ memory for its butterfly computation, and the multi-scalar multiplication (MSM), which maintains $\mathcal{O}(N)$ bucket accumulators (with the optimal window size $c = \log N$, yielding $2^c = N$ buckets).

Scribe [2] introduced the *read-write streaming model*, showing that HyperPlonk [3] can be proved with $\mathcal{O}(\log N)$ RAM by streaming over $\mathcal{O}(N)$ external storage. This suggests a useful memory-saving template, but it does not preserve the verifier contract of deployed Groth16 systems: concrete proof size and verification cost depend on the polynomial-commitment scheme and verifier used by the underlying proof system. The question for Groth16 is therefore:

> *Can the Groth16 prover operate in the read-write streaming model, preserving constant-size proofs and constant-pairing verification while using only $\mathcal{O}(\log N)$ prove-phase random-access memory once its inputs are materialized as streams?*

Scope of the memory claim. All asymptotic and measured memory bounds in this paper refer to the prove worker after CRS, QAP/R1CS, and witness streams have been materialized as private prover-controlled storage. Setup, witness generation, stream materialization, cleanup, and persistent storage footprint are accounted for separately and are not claimed to be $\mathcal{O}(\log N)$. Throughout, N denotes the padded QAP/NTT domain size unless stated otherwise.

The main obstacle is the NTT. HyperPlonk's sumcheck folds adjacent pairs (stride 1), streaming naturally; Groth16 relies on butterfly stages with stride $2^j \gg \log N$.

1.1. Our Contributions

Sluice answers this question. It is a streaming Groth16 prover for the read-write streaming model, with the following contributions.

1. Split-Butterfly-Merge (SBM): streaming NTT with $\mathcal{O}(\log N)$ memory (Section 3). We give an explicit algorithm that performs the radix-2 NTT in the read-write streaming model. For each large-stride butterfly stage, the algorithm partitions the data into two streams by the relevant bit position, performs all butterfly operations in a single sequential pass over the paired streams, and merges the results back. The SBM algorithm computes an N-point

NTT using $\mathcal{O}(\log N)$ random-access memory, $\mathcal{O}(N \log N)$ field operations and total I/O, and $\mathcal{O}(\log N)$ sequential passes over external storage. We also give a streaming bit-reversal permutation with the same resource bounds. The construction uses stage-wise bit partitioning rather than a new external-memory lower bound. We compare it with the natural generic sort/permutation streaming baseline in Table 1.

2. Sluice: streaming Groth16 prover construction (Section 5). Combining SBM with the streaming multi-scalar multiplication technique of Scribe [2] (chunked Pippenger) and a streaming sparse matrix-vector multiplication for R1CS evaluation, we construct a Groth16 prover in the read-write streaming model. It uses $\mathcal{O}(\log N)$ prove-phase random-access memory once the CRS, matrices, and witness are supplied as private streams. Given the same randomness, it produces the same proof as the standard Groth16 prover, and it preserves Groth16's constant proof size, constant-pairing verification, and public-input accumulation. The verifier is unchanged; witness generation and streaming setup are outside this prove-phase memory bound.

3. Security via identity reduction (Section 6). Since Sluice shares the same setup, the same verification equation, and produces the same proof distribution as standard Groth16, we show that its knowledge soundness and zero-knowledge properties are inherited via the identity reduction—no rewinding, no simulation overhead, and no strengthening of the underlying assumptions.

4. Complexity analysis and comparison (Section 7). We compare Sluice with the standard Groth16 prover and with Scribe. With MSM chunk size $B_{\mathrm{MSM}} = \Theta(\log N)$, Sluice achieves $\mathcal{O}(\log N)$ prove-phase memory (vs. $\mathcal{O}(N)$ for Groth16) with a $\mathcal{O}(\log N / \log\log N)$ asymptotic group-operation overhead, while maintaining constant proof size and constant-pairing verification and the same verifier-facing interface as Groth16. The prototype uses a fixed $w = 8$ MSM window, so its measurements validate the memory/proof-compatibility trade-off rather than the optimized asymptotic MSM schedule.

Section 8 presents an implementation and experimental evaluation of the resulting memory/I/O trade-off.

1.2. Related Work

External-memory and streaming algorithms. The FFT with limited memory has a long history. Bailey's four-step FFT [4] and Stockham/autosort-style layouts regularize memory traffic but still operate over addressable arrays or block transfers; Hong and Kung [5] proved an $\Omega(N \log N / \log M)$ I/O lower bound; Aggarwal and Vitter [6] extended this to external-memory sorting; and cache-oblivious algorithms [7] achieve optimal cache complexity but still assume a memory hierarchy with randomly addressable blocks. SBM targets a stricter interface: every large vector is consumed and produced sequentially, with only $\mathcal{O}(\log N)$ live state. The contribution is not a new FFT lower bound but a bit-partition primitive that realizes Groth16's non-unit-stride NTT stages by sequential scans, giving $\mathcal{O}(N \log N)$ I/O within $\mathcal{O}(\log\log N)$ of the Hong–Kung bound.

QAP-based SNARKs. QAPs were introduced by Gennaro *et al.* [8] and made practical by Pinocchio [9]. Groth16 [1] achieves optimal proof size and verification for pairing-based SNARKs; subsequent work addressed extensions [10], [11] and multi-party setup [12]. Our work preserves Groth16's cryptographic construction unchanged.

Alternative proof systems. Sumcheck-based systems [13], [14], [15], universal-SRS schemes (PLONK [16], Marlin [17], HyperPlonk [3]), and STARKs [18] offer different trade-offs. Our streaming NTT technique is orthogonal and may benefit other NTT-reliant proof systems.

Memory-efficient SNARKs. Scribe [2] formalizes the read-write streaming model and applies it to HyperPlonk, whose stride-1 sumcheck streams naturally. Hobbit [19] constructs a new space-efficient SNARK that achieves optimal prover time with reduced memory by redesigning the underlying polynomial IOP. These systems address different goals. Our contribution is narrower: rather than designing a new proof system, we supply the NTT primitive missing from Scribe's model and retrofit the *existing* Groth16 prover with read-write streaming while preserving verifier-facing compatibility with deployed Groth16 verifiers.

Practical deployments. Groth16 is deployed in Zerocash [20] and zk-rollups [21]; production provers (bellman [22], arkworks [23], rapidsnark [24]) are heavily optimized and retain linear-size working-state components. We do not claim superiority over optimized production provers. Sluice is verifier-compatible with these systems, but our prototype targets RAM-limited, storage-rich hosts with trusted local storage, or controlled-cloud settings where snapshots, swap, and storage recovery are under the prover operator's control. Its evaluation uses synthetic scaling instances through $N = 2^{25}$, a MiMC-style Merkle structured workload, and an arkworks baseline through $N = 2^{23}$; production workloads and broader prover baselines remain future work.

Why Groth16 persists. Despite universal-SRS and transparent proof systems, Groth16 remains deployed because BN-254 pairing checks are widely supported and existing verifier contracts are costly to redeploy and re-audit. Sluice addresses prover memory while preserving compatibility with this infrastructure.

1.3. Paper Organization

Section 2 covers preliminaries; Section 3 presents SBM; Section 4 describes streaming MSM; Section 5 assembles Sluice; Sections 6 and 7 provide security and complexity analysis; Section 8 reports experiments.

2. Preliminaries

2.1. Notation

We write $\mathbb{F}$ for a finite field of prime order p, with $\lambda = |p|$ denoting the bit length of p. Vectors are written

in boldface ($\mathbf{a} = (a_0, \ldots, a_{N-1})$). For a positive integer N, we write $[N] = \{0, 1, \ldots, N-1\}$. We set $n = \log_2 N$ when N is a power of two. Since $\mathbb{F}$ has prime order p, the scalar field is also $\mathbb{F}$; we use $\mathbb{F}$ for QAP values and scalar sampling.

For $x \in [N]$, we write $\mathsf{bit}_j(x)$ for the j-th bit of x (least-significant bit is bit 0). The *bit-reversal* of x over n bits is $\mathsf{bitrev}_n(x) = \sum_{j=0}^{n-1} \mathsf{bit}_j(x) \cdot 2^{n-1-j}$.

We use standard Landau notation: $\mathcal{O}(\cdot)$, $\Omega(\cdot)$, $\Theta(\cdot)$. Logarithms are base 2 unless stated otherwise.

2.2. Bilinear Groups

Definition 2.1 (Bilinear group generator)**.** A *bilinear group generator* $\mathcal{G}(1^\lambda)$ outputs $(p, \mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T, e, g, h)$ where:

- $\mathbb{G}_1$, $\mathbb{G}_2$, $\mathbb{G}_T$ are cyclic groups of prime order p, with $|p| = \lambda$;
- g generates $\mathbb{G}_1$ and h generates $\mathbb{G}_2$;
- $e\colon \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is a non-degenerate bilinear map (pairing): $e(g^a, h^b) = e(g, h)^{ab}$ for all $a, b \in \mathbb{F}$.

We use *Type III* pairings, where no efficiently computable isomorphism between $\mathbb{G}_1$ and $\mathbb{G}_2$ is assumed.

Following Groth [1], we use *bracket notation*: for $a \in \mathbb{F}$, we write $[a]_1 = a \cdot g \in \mathbb{G}_1$ and $[a]_2 = a \cdot h \in \mathbb{G}_2$. The pairing satisfies $e([a]_1, [b]_2) = [ab]_T$ for all $a, b \in \mathbb{F}$.

2.3. Quadratic Arithmetic Programs

Definition 2.2 (R1CS / QAP)**.** A *rank-1 constraint system* of size d over $\mathbb{F}$ consists of three $d \times (m{+}1)$ matrices U, V, W. An assignment $\mathbf{a} = (1, a_1, \ldots, a_m)$ satisfies the system if $(U\mathbf{a}) \circ (V\mathbf{a}) = W\mathbf{a}$. The entries $a_1, \ldots, a_\ell$ form the *statement*; the rest form the *witness*.

A *quadratic arithmetic program* (QAP) encodes this R1CS over a power-of-two domain $N = 2^{\lceil \log d \rceil}$. Rows $d, \ldots, N{-}1$ are padded with zero constraints, i.e., $U_{k,i} = V_{k,i} = W_{k,i} = 0$ for every padded row k and wire i. The polynomials u_i, v_i, w_i of degree $<N$ are then uniquely defined by interpolation over the N-th roots of unity: $u_i(\omega^k) = U_{k,i}$, etc. The target polynomial is $t(X) = X^N{-}1$. The assignment satisfies the QAP iff $t(X)$ divides $(\sum_i a_i\, u_i) \cdot (\sum_i a_i\, v_i) - \sum_i a_i\, w_i$, yielding the quotient $h(X)$ of degree $\leq N{-}2$.

2.4. The Groth16 zk-SNARK

We recall the Groth16 construction [1].

Setup. The setup algorithm $(\sigma, \tau_{\mathrm{trap}}) \leftarrow \mathsf{Setup}(R)$ samples $\alpha, \beta, \tau \leftarrow \mathbb{F}$ and $\gamma, \delta \leftarrow \mathbb{F}^*$, equivalently resampling if either denominator is zero or if $t(\tau) = 0$ (i.e., τ lies in the evaluation domain). Here $\tau_{\mathrm{trap}} = (\alpha, \beta, \gamma, \delta, \tau)$ and τ is the QAP evaluation point. It outputs a CRS σ consisting of group elements:

$$
\begin{aligned}
&[\alpha]_1,\ [\beta]_1,\ [\delta]_1,\ \{[\tau^j]_1\}_{j=0}^{N-1},\\
&\{[u_i(\tau)]_1\}_{i=0}^{m},\ \{[v_i(\tau)]_1\}_{i=0}^{m},\\
&\Big\{\big[\tfrac{\beta u_i + \alpha v_i + w_i}{\gamma}\big]_1\Big\}_{i=0}^{\ell},\ \Big\{\big[\tfrac{\beta u_i + \alpha v_i + w_i}{\delta}\big]_1\Big\}_{i=\ell+1}^{m},\\
&\Big\{\big[\tfrac{\tau^j t(\tau)}{\delta}\big]_1\Big\}_{j=0}^{N-2},\ [\beta]_2,\ [\gamma]_2,\ [\delta]_2,\ \{[v_i(\tau)]_2\}_{i=0}^{m}.
\end{aligned}
$$

The *verifying key* contains $[\alpha]_1$, $[\beta]_2$, $[\gamma]_2$, $[\delta]_2$, and $\{[(\beta u_i(\tau) + \alpha v_i(\tau) + w_i(\tau))/\gamma]_1\}_{i=0}^{\ell}$.

Prover. Given a statement $\phi = (a_1, \ldots, a_\ell)$ and witness $w = (a_{\ell+1}, \ldots, a_m)$, the prover samples $r, s \leftarrow \mathbb{F}$ and computes:

$$A = \alpha + \textstyle\sum_{i=0}^{m} a_i\, u_i(\tau) + r\delta, \tag{1}$$

$$B = \beta + \textstyle\sum_{i=0}^{m} a_i\, v_i(\tau) + s\delta, \tag{2}$$

$$C = \tfrac{1}{\delta}\big(\textstyle\sum_{i=\ell+1}^{m} a_i(\beta u_i(\tau) + \alpha v_i(\tau) + w_i(\tau)) + h(\tau)\, t(\tau)\big) + As + rB - rs\delta. \tag{3}$$

The proof is $\pi = ([A]_1, [B]_2, [C]_1)$.

Verifier. Given a statement ϕ and proof $\pi = ([A]_1, [B]_2, [C]_1)$, the verifier computes the public-input accumulator $[L_\phi]_1 = \sum_{i=0}^{\ell} a_i \cdot [(\beta u_i(\tau) + \alpha v_i(\tau) + w_i(\tau))/\gamma]_1$ and checks:

$$e([A]_1, [B]_2) \;=\; e([\alpha]_1, [\beta]_2) \cdot e([L_\phi]_1, [\gamma]_2) \cdot e([C]_1, [\delta]_2). \tag{4}$$

Theorem 2.3 (Groth16 [1, Theorems 1–3])**.** *Groth16 is a publicly verifiable NIZK argument of knowledge with perfect completeness, computational knowledge soundness (in the generic group model), and perfect zero-knowledge.*

2.5. The Number Theoretic Transform

The *Number Theoretic Transform* (NTT) computes the evaluation of a polynomial over a multiplicative subgroup of $\mathbb{F}$.

Definition 2.4 (NTT)**.** Let $\omega \in \mathbb{F}$ be a primitive N-th root of unity with $N = 2^n$. The NTT of a vector $\mathbf{f} = (f_0, \ldots, f_{N-1}) \in \mathbb{F}^N$ is $\hat{\mathbf{f}} = (\hat{f}_0, \ldots, \hat{f}_{N-1})$ where $\hat{f}_k = \sum_{j=0}^{N-1} f_j\, \omega^{jk}$ for $k \in [N]$. The inverse NTT (iNTT) recovers $\mathbf{f}$ from $\hat{\mathbf{f}}$ using the inverse root ω^{-1} and a normalization factor of N^{-1}.

The standard radix-2 Cooley–Tukey algorithm [25] computes the NTT in $\mathcal{O}(N \log N)$ field operations using $\mathcal{O}(N)$ memory. The algorithm consists of $n = \log N$ stages; at stage j ($0 \leq j < n$), each butterfly operation pairs elements at positions p and $p \oplus 2^j$ (stride 2^j) and applies a twiddle-factor multiplication. For small strides ($2^j \ll N$), the two operands are nearby in memory; for large strides ($2^j \gg \log N$), they are separated by up to $N/2$ positions, preventing sequential access.

The Groth16 prover uses the NTT and its inverse in three roles: (i) evaluating the aggregate QAP polynomials

$F_U(X), F_V(X), F_W(X)$ at the roots of unity (via the R1CS-to-QAP interpolation), (ii) evaluating these polynomials on a coset for the quotient computation, and (iii) interpolating the quotient polynomial $h(X)$ back to coefficient form. In total, the $h(X)$ computation requires 7 NTT/iNTT invocations (3 inverse transforms for coefficient recovery, 3 forward transforms for coset evaluation, and 1 inverse transform for the quotient). Reducing the memory of each NTT invocation to $\mathcal{O}(\log N)$ while preserving the $\mathcal{O}(N \log N)$ operation count is the main technical challenge addressed in Section 3.

2.6. The Read-Write Streaming Model

We adopt the *read-write streaming model* of Scribe [2].

Definition 2.5 (Read-write streaming [2, Def. 2.1])**.** A read-write streaming algorithm has $m_{\mathcal{A}}$ *registers* (random-access memory) and sequential-access *streams* supporting $\mathsf{read}()$, $\mathsf{write}(x)$, and $\mathsf{restart}()$. Complexity is measured by: RAM ($m_{\mathcal{A}}$, in field/group elements), *streaming space* (total intermediate-stream size), *I/O cost* (total reads + writes), and *passes* (sequential scans; a restart counts as a new pass).

Intuition. The model captures algorithms that process large data via sequential scans rather than random access—analogous to reading a file from start to finish, rather than seeking to arbitrary positions. A stream can be "rewound" ($\mathsf{restart}$) to be read again from the beginning, but random seeking within a stream is not permitted. This model maps naturally to disk-backed computation: the streams correspond to files on an SSD or HDD, while the registers correspond to variables held in RAM. The key constraint is that the number of registers is *sublinear* in the input size (in our case, $\mathcal{O}(\log N)$), forcing the algorithm to perform its computation through carefully orchestrated sequential passes.

Lemma 2.6 (Composition [2, Lemma 2.1])**.** *Sequential composition $\mathcal{B} \circ \mathcal{A}$ uses $\max(m_A, m_B)$ registers and $\max(s_A, s_B)$ streaming space when intermediate streams are reused.*

The composition lemma is critical for Sluice: it allows us to compose the SBM NTT (Section 3), the streaming MSM (Section 4), and the streaming R1CS evaluation (Section 5) into a single prover pipeline whose memory is the *maximum* of the individual components—not their sum. Since each component uses $\mathcal{O}(\log N)$ registers, the composed prover also uses $\mathcal{O}(\log N)$ registers.

3. Streaming NTT via Split-Butterfly-Merge

We present a read-write streaming NTT using $\mathcal{O}(\log N)$ RAM, $\mathcal{O}(N \log N)$ I/O, and $\mathcal{O}(\log N)$ passes. Let $N = 2^n$ and $\omega \in \mathbb{F}$ a primitive N-th root of unity.

3.1. The Challenge: Butterfly Patterns vs. Streaming

The radix-2 DIT NTT has $n = \log N$ stages; stage j pairs elements at positions p and $p \oplus 2^j$ (stride 2^j). Small-stride stages ($2^{j+1} \le c \log N$) fit in a streaming buffer, but large-stride stages cannot: the two operands are too far apart for a single sequential pass. Scribe [2] sidesteps this via HyperPlonk's [3] stride-1 sumcheck, but Groth16 fundamentally requires the NTT with exponentially growing strides. A generic read-write approach can sort or permute each stage by its butterfly-pair key, but this repeats a streaming radix sort across all $\log N$ stages.

Method	RAM	Total I/O	Passes
In-memory NTT	$\mathcal{O}(N)$	$\mathcal{O}(N \log N)$	RAM-resident
Generic sort/permute	$\mathcal{O}(\log N)$	$\mathcal{O}\big(N \log^2 N\big)$	$\mathcal{O}\big(\log^2 N\big)$
SBM	$\mathcal{O}(\log N)$	$\mathcal{O}(N \log N)$	$\mathcal{O}(\log N)$

TABLE 1. NTT IMPLEMENTATION CHOICES UNDER A READ-WRITE STREAMING INTERFACE. THE GENERIC ROW ASSUMES A PER-STAGE READ-WRITE STREAMING SORT/PERMUTATION (E.G., RADIX SORT BY THE STAGE-PAIR KEY) TO ALIGN BUTTERFLY PARTNERS AND RESTORE ORDER, PAYING AN EXTRA $\log N$ FACTOR. SBM USES A BIT PARTITION AND MERGE WITH A CONSTANT NUMBER OF STREAMS PER STAGE.

3.2. Bit-Partitioning and Butterfly Pairing

High-level intuition. Consider a stage-j butterfly that pairs positions p and $p \oplus 2^j$. These two positions differ only in bit j of their binary representation: one has bit $j = 0$, the other has bit $j = 1$. If we *partition* all N positions into two half-size streams according to bit j—the "0-stream" and the "1-stream"—then every butterfly pair $(p,\, p \oplus 2^j)$ maps to the *same index* within their respective streams. This is because removing bit j from both p and $p \oplus 2^j$ yields the same $(n-1)$-bit integer. Consequently, we can perform all $N/2$ butterfly operations for stage j by reading the two streams in lockstep, applying the butterfly to each aligned pair, and writing the results back—all in a single sequential pass with $\mathcal{O}(1)$ memory per butterfly. The challenge then reduces to implementing the partition (Split) and the subsequent reassembly (Merge) as streaming operations, which we formalize below.

Our key observation is that partitioning the data by bit j of the position index aligns butterfly partners at the same relative index in two separate streams.

We use the standard iterative radix-2 DIT convention: after the initial bit-reversal and every previous stage, the stream is in the array order expected by the next DIT stage. Stage j applies butterflies to positions p and $p \oplus 2^j$ with $\mathsf{bit}_j(p) = 0$ using the twiddle defined below.

Definition 3.1 (Bit-j partition)**.** For $0 \le j < n$, define

$$\mathcal{A}_j = \{\, p \in [N] : \mathsf{bit}_j(p) = 0 \,\},$$
$$\mathcal{B}_j = \{\, p \in [N] : \mathsf{bit}_j(p) = 1 \,\}.$$

Let $\sigma_j : \mathcal{A}_j \to [N/2]$ and $\tau_j : \mathcal{B}_j \to [N/2]$ be the order-preserving bijections that map each set to $\{0, \ldots, N/2 - 1\}$ while preserving the natural ordering.

Lemma 3.2 (Pairing lemma)**.** *For every $p \in \mathcal{A}_j$, its butterfly partner $p \oplus 2^j$ lies in $\mathcal{B}_j$ and satisfies $\sigma_j(p) = \tau_j(p \oplus 2^j)$.*

Proof. Deleting bit j from the binary representation of p yields the same $(n-1)$-bit string for both $\sigma_j(p)$ and $\tau_j(p \oplus 2^j)$. See Section A for details. □

Each butterfly at stage j uses a *twiddle factor*:

$$\omega_q = \omega^{(q \bmod 2^j)\cdot 2^{n-j-1}}, \qquad 0 \le q < N/2. \tag{5}$$

Lemma 3.3 (Twiddle factor identity)**.** *In the DIT NTT, the twiddle factor for the butterfly at positions $p \in \mathcal{A}_j$ and $p \oplus 2^j \in \mathcal{B}_j$ is $\omega^{(p \bmod 2^j)\cdot 2^{n-j-1}}$. If $q = \sigma_j(p)$ is the relative index, then $p \bmod 2^j = q \bmod 2^j$, so the twiddle factor equals ω_q as defined in (5).*

Proof. The lower j bits of p are $(b_{j-1}, \ldots, b_0)$, which form the lower j bits of the rank $q = \sigma_j(p)$ since σ_j merely removes bit j and shifts higher bits down by one position. □

Crucially, the twiddle factors can be computed on-the-fly with $\mathcal{O}(1)$ registers, avoiding storage of all $N/2$ values.

Lemma 3.4 (On-the-fly twiddle computation)**.** *Define the step constant $W_j = \omega^{2^{n-j-1}}$ and the sequence $\{\omega_{\text{cur}}^{(q)}\}_{q \ge 0}$ by*

$$\omega_{\text{cur}}^{(0)} = 1, \quad \omega_{\text{cur}}^{(q+1)} = \begin{cases} \omega_{\text{cur}}^{(q)} \cdot W_j & (q+1) \bmod 2^j \neq 0, \\ 1 & (q+1) \bmod 2^j = 0. \end{cases}$$

Then for all $0 \le q < N/2$, $\omega_{\text{cur}}^{(q)} = \omega_q$ as defined in (5). The computation uses exactly two field-element registers (ω_{cur} and the constant W_j) and one $\mathcal{O}(\log N)$-bit counter.

Proof. After $r = q \bmod 2^j$ multiplications by W_j since the last reset, $\omega_{\text{cur}}^{(q)} = W_j^r = \omega^{r \cdot 2^{n-j-1}} = \omega_q$ by Theorem 3.3. See Section A for the full argument. □

3.3. The Split-Butterfly-Merge Algorithm

Three subroutines process one large-stride FFT stage.

Split. Given an input stream I of N elements and an index j, the Split subroutine performs one sequential pass over I. For each position p, it reads the element x_p and writes it to $\mathcal{S}_A$ if $\mathsf{bit}_j(p) = 0$, or to $\mathcal{S}_B$ if $\mathsf{bit}_j(p) = 1$. The only state is a position counter ($\mathcal{O}(\log N)$ bits).

Paired Butterfly. Given streams $\mathcal{S}_A, \mathcal{S}_B$ (each of length $N/2$), the PairedBfly subroutine reads one element from each stream in lockstep. For the q-th pair, using the twiddle factor ω_q from (5), it computes

$$y_q^{(\text{top})} = a_q + \omega_q \cdot b_q, \tag{6}$$

$$y_q^{(\text{bot})} = a_q - \omega_q \cdot b_q, \tag{7}$$

and writes $y_q^{(\text{top})}$ to output stream O_A and $y_q^{(\text{bot})}$ to O_B.

The twiddle factor (5) is maintained on-the-fly: a register ω_{cur} is initialized to 1 and multiplied by $\omega^{2^{n-j-1}}$ at each step, resetting to 1 every 2^j steps. The total state is two field elements (ω_{cur} and the step constant) plus an $\mathcal{O}(\log N)$-bit counter.

Merge. Merge reconstructs natural order: for output position p, read from O_A if $\mathsf{bit}_j(p) = 0$, else from O_B. Because these positions form contiguous blocks of size 2^j, both streams are read sequentially; scanning output positions alternates runs of 2^j items from O_A and O_B.

Theorem 3.5 (SBM correctness)**.** *For stage j of the DIT NTT with stride 2^j, the composition* Merge ∘ PairedBfly ∘ Split *applied to an input in natural order produces the same output as the standard in-place butterfly computation for stage j.*

Proof sketch. By Theorems 3.2 and 3.3, Split pairs each butterfly's operands at the same relative index q in the two sub-streams, and PairedBfly applies the correct twiddle ω_q. Merge reads O_A and O_B sequentially because positions with $\mathsf{bit}_j(p) = 0$ (resp. 1) appear in contiguous blocks of size 2^j. See Section A for the complete proof. □

3.4. Streaming Bit-Reversal Permutation

The DIT NTT requires bit-reversed input. The bit-reversal $\mathsf{bitrev}(p) = (b_0, \ldots, b_{n-1})$ decomposes into $\lfloor n/2 \rfloor$ independent transpositions (swap bit k with $n-1-k$), each realized by a streaming BitSwap.

Definition 3.6 (Streaming bit-swap)**.** $\mathsf{BitSwap}(I, i, j)$ permutes a stream by swapping bits i and j in every index, using two passes: (1) *Split*—route each element to one of four streams $S_{b_i b_j}$ based on $(\mathsf{bit}_i(p), \mathsf{bit}_j(p))$; (2) *Cross-merge*—for output position q with $(\mathsf{bit}_i(q), \mathsf{bit}_j(q))$, read from $S_{\mathsf{bit}_j(q)\,\mathsf{bit}_i(q)}$ (subscript bits swapped).

Lemma 3.7 (Bit-swap correctness)**.** $\mathsf{BitSwap}(I, i, j)$ *correctly computes the permutation that swaps bits i and j in every position index, using 2 passes, $\mathcal{O}(\log N)$ memory, and 4 intermediate streams.*

Proof. Each element x_p is routed to $S_{b_i b_j}$; the cross-merge reads from $S_{b_i b_j}$ at position p' where bits i, j are swapped. Order-preservation within each sub-stream ensures correctness. See Section A. □

Theorem 3.8 (Streaming bit-reversal)**.** *The bit-reversal permutation on $N = 2^n$ elements can be computed in the read-write streaming model using $\mathcal{O}(\log N)$ memory, $2\lfloor n/2 \rfloor \le n$ passes, and 4 intermediate streams of total size $\mathcal{O}(N)$.*

Proof. Compose $\lfloor n/2 \rfloor$ bit-swaps, each using 2 passes. Intermediate streams are recycled between successive swaps by the composition lemma of [2]. See Section A. □

3.5. Complete RW-NTT Construction

The algorithm is formalized in Algorithm 1. Small-stride stages ($2^{j+1} \le c \log N$) fit in the streaming buffer and are handled in-memory; large-stride stages use the three-pass SBM decomposition from Section 3.3. $\mathsf{DirectBfly}_j$ buffers one block of 2^{j+1} consecutive elements, applies the standard stage-j butterflies inside the block, and writes the block back

Algorithm 1: RW-NTT: Streaming NTT via Split-Butterfly-Merge

Input: Input stream I of $N = 2^n$ field elements
Output: Output stream O containing $\mathsf{NTT}(I)$

```
/* Phase 1: Streaming bit-reversal
   (Theorem 3.8)                          */
```
$I' \leftarrow \mathsf{bitrev}(I)$ via $\lfloor n/2 \rfloor$ streaming BitSwaps

```
/* Phase 2: Stage loop                    */
```
for $j \leftarrow 0$ **to** $n-1$ **do**
  if $2^{j+1} \le c \log N$ **then**
    $I' \leftarrow \mathsf{DirectBfly}_j(I')$ // buffer 2^{j+1} elements in RAM
  else
    $(\mathcal{S}_A, \mathcal{S}_B) \leftarrow \mathsf{Split}_j(I')$ // partition by $\mathsf{bit}_j(p)$
    $(O_A, O_B) \leftarrow \mathsf{PairedBfly}_j(\mathcal{S}_A, \mathcal{S}_B)$ // lockstep butterfly
    $I' \leftarrow \mathsf{Merge}_j(O_A, O_B)$ // restore natural order

return I'

in natural order. Its memory is $\mathcal{O}\big(2^{j+1}\big) \le \mathcal{O}(\log N)$ by the small-stage condition.

Theorem 3.9 (RW-NTT correctness)**.** RW-NTT *computes the same output as the standard N-point NTT.*

Proof. Bit-reversal is correct by Theorem 3.8; each subsequent stage is correct by either direct in-memory computation (small stride) or Theorem 3.5 (large stride), and stages compose because each preserves natural order. See Section A. □

3.6. Complexity Analysis

Theorem 3.10 (RW-NTT complexity)**.** *The* RW-NTT *algorithm computes the N-point NTT with the following complexity:*

1) ***Random-access memory:*** *$\mathcal{O}(\log N)$ field elements.*
2) ***Streaming space:*** *$\mathcal{O}(N)$ field elements (across at most 4 intermediate streams, each of size at most N).*
3) ***Passes:*** *At most $4n+1 = \mathcal{O}(\log N)$, where $n = \log N$.*
4) ***Total I/O:*** *$\mathcal{O}(N \log N)$ read/write operations.*
5) ***Field operations:*** *$\mathcal{O}(N \log N)$. The butterfly arithmetic matches the standard NTT; on-the-fly twiddle maintenance adds only a constant number of field operations per butterfly and no precomputed twiddle table.*

Proof. Bit-reversal contributes n passes; each of the $\mathcal{O}(n)$ large-stride stages contributes 3 passes (Split, Butterfly, Merge), giving $\le 4n + \mathcal{O}(1)$ passes total. Memory is dominated by $\mathcal{O}(\log N)$ (position counters and field-element registers), with streams reused across stages via the composition lemma of [2]. Each stage performs $\mathcal{O}(N)$ butterfly and twiddle-update operations, so the total field-operation count remains $\mathcal{O}(N \log N)$. The detailed accounting appears in Section A. □

Remark 3.11 (Near-optimality)**.** The Hong–Kung [5] lower bound gives $\Omega(N \log N / \log\log N)$ I/O for cache $M = \mathcal{O}(\log N)$. Our $\mathcal{O}(N \log N)$ exceeds this by at most an $\mathcal{O}(\log\log N)$ factor (≤ 5 for $N \le 2^{32}$).

Remark 3.12 (Inverse NTT)**.** The inverse NTT (iNTT) is computed identically, replacing ω by ω^{-1} and scaling the final output by N^{-1}. This multiplication is fused into the final output write of the inverse transform and therefore does not require an additional streaming pass. The streaming structure is unchanged, so RW-NTT handles both forward and inverse transforms with the same complexity.

4. Streaming Multi-Scalar Multiplication

Each Groth16 proof element is a multi-scalar multiplication (MSM) $\sum_i a_i \cdot G_i$. We adapt the chunked Pippenger technique of Scribe [2] for the read-write streaming model.

4.1. Pippenger's Algorithm and Its Memory Bottleneck

Pippenger's algorithm [26] computes an MSM $\sum_{i=1}^{N} a_i \cdot G_i$ as follows. Each λ-bit scalar a_i is decomposed into $\lceil \lambda/c \rceil$ windows of c bits. For each window, the base points are sorted into 2^c buckets according to the c-bit value, each bucket is summed, and the bucket sums are aggregated via a descending-weight scan. The total cost is $\mathcal{O}(\lambda N/c)$ group additions; the optimal window $c = \lfloor \log_2 N \rfloor$ gives $\mathcal{O}(\lambda N / \log N)$ group operations.

However, the optimal window requires $2^c = N$ bucket accumulators, each a group element, resulting in $\Theta(N)$ random-access memory. This is a dominant memory cost in the Groth16 prover: at $N = 2^{23}$ on BN-254 (64 bytes per $\mathbb{G}_1$ element), the buckets alone occupy ≈ 512 MiB. Using a smaller window $c < \log N$ reduces memory to 2^c but increases the operation count by a factor of $\log N / c$.

4.2. Streaming MSM via Chunked Pippenger

Scribe [2] observes that Pippenger's algorithm can be applied "in chunks" to obtain a streaming MSM with tunable memory.

Definition 4.1 (Chunked streaming MSM)**.** Let B_{MSM} be the MSM chunk-size parameter. The *chunked Pippenger MSM* proceeds as follows:

1) Initialize an accumulator $\mathsf{ans} \leftarrow \mathcal{O}$ (the identity element).
2) Stream through the input in chunks of B_{MSM} scalar/group-element pairs: for $k = 0, 1, \ldots, \lceil N/B_{\mathrm{MSM}} \rceil - 1$, read the k-th chunk into random-access memory.
3) Apply Pippenger's algorithm to the chunk in memory, obtaining its partial MSM P_k.
4) Update $\mathsf{ans} \leftarrow \mathsf{ans} + P_k$.

5) After all chunks, output ans.

Theorem 4.2 (Streaming MSM complexity)**.** *The chunked Pippenger MSM with chunk size B_{MSM} computes an N-element MSM in a group of order $|p| = \lambda$ bits with:*

1) ***Random-access memory:*** *$\mathcal{O}(B_{\mathrm{MSM}})$ group elements.*
2) ***Group operations:*** *$\mathcal{O}(\lambda N/\log B_{\mathrm{MSM}})$.*
3) ***Passes:*** *1 (single sequential pass over the input).*
4) ***Streaming space:*** *$\mathcal{O}(N)$ (input streams for scalars and group elements).*

Proof. Each chunk uses Pippenger with window $c = \log B_{\mathrm{MSM}}$: per-chunk cost $\mathcal{O}(\lambda B_{\mathrm{MSM}}/\log B_{\mathrm{MSM}})$; over $\lceil N/B_{\mathrm{MSM}} \rceil$ chunks, the total is $\mathcal{O}(\lambda N/\log B_{\mathrm{MSM}})$. Memory holds B_{MSM} pairs plus bucket accumulators. Input is consumed in one sequential pass. □

Remark 4.3 (Choice of B_{MSM} for Sluice)**.** Setting $B_{\mathrm{MSM}} = \mathcal{O}(\log N)$ yields $\mathcal{O}(\log N)$ random-access memory and $\mathcal{O}(\lambda N/\log\log N)$ group operations. Since λ is fixed (e.g., $\lambda = 254$ for BN-254 and about 255 bits for BLS12-381), this is $\mathcal{O}(N/\log\log N)$ group operations, which is only a $\mathcal{O}(\log N/\log\log N)$ factor slower than the standard Pippenger bound of $\mathcal{O}(\lambda N/\log N)$.

4.3. Application to Groth16

The Groth16 prover requires five MSMs: three assignment MSMs of size $m+1$, one witness-key MSM of size $m-\ell$, and one quotient-key MSM of size $N-1$. Thus the total MSM input length is $\mathcal{O}(m+N)$, where m is the number of nonconstant wires. Scalars come from the QAP evaluation streams; CRS group elements are stored as contiguous streams on external storage.

Proposition 4.4 (Streaming MSM for Groth16)**.** *With $B_{\mathrm{MSM}} = \mathcal{O}(\log N)$, the Groth16 MSM phase uses $\mathcal{O}(\log N)$ RAM, $\mathcal{O}(\lambda(m+N)/\log\log N)$ group operations, and one sequential pass over each CRS stream.*

Proof. Direct application of Theorem 4.2 with $B_{\mathrm{MSM}} = \mathcal{O}(\log N)$ to MSM streams whose total length is $\mathcal{O}(m+N)$. □

Lemma 4.5 (Chunked MSM correctness)**.** *The correctness of chunked Pippenger follows from the linearity of the group operation: for any partition of the index set $[N]$ into disjoint chunks $I_0, I_1, \ldots, I_{K-1}$ with $K = \lceil N/B_{\mathrm{MSM}} \rceil$,*

$$\sum_{i=1}^{N} a_i \cdot G_i = \sum_{k=0}^{K-1} \Big(\sum_{i \in I_k} a_i \cdot G_i \Big) = \sum_{k=0}^{K-1} P_k.$$

Each partial MSM P_k is computed exactly (using Pippenger's algorithm in RAM), and the accumulation ans $\leftarrow$ ans$+P_k$ *preserves this equality. No approximation or rounding is involved.*

Practical considerations. In our implementation, the chunk size B_{MSM} is set to match the Pippenger window size w via $B_{\mathrm{MSM}} = 2^w$ for efficient bucket addressing. At $w = 8$ this gives $B_{\mathrm{MSM}} = 256$—much larger than $\log N$ for practical padded domain sizes ($\log N \le 25$)—but still provides substantial memory savings compared to standard Pippenger's millions of buckets at our largest comparison point. The asymptotic theorem uses $B_{\mathrm{MSM}} = \Theta(\log N)$ to optimize the group-operation bound; the prototype fixes $w = 8$ as a conservative constant-memory engineering point. This instantiates the same streaming MSM design, but not the asymptotically tuned MSM schedule. With fixed $w = 8$, the implemented MSM uses $\mathcal{O}(\lambda N/w) = \mathcal{O}(\lambda N)$ group operations, i.e., a $\log N/w$ window-overhead factor relative to standard Pippenger's $\mathcal{O}(\lambda N/\log N)$ bound. Thus the experiments validate the memory/I/O design and proof compatibility of the fixed-window prototype, not the optimized asymptotic MSM schedule. The choice of $w = 8$ is empirically validated by the appendix component measurements in Figure 2(b). Increasing w beyond 8 causes the bucket array to exceed the L3 cache, triggering severe performance degradation.

Remark 4.6 (CRS storage layout)**.** The five MSM CRS arrays are stored on disk as contiguous streams of serialized group elements: four $\mathbb{G}_1$-streams (for $[A]_1$, auxiliary $[B]_1$, the witness part of $[C]_1$, and the quotient part of $[C]_1$) and one $\mathbb{G}_2$-stream (for $[B]_2$). Each element is stored in uncompressed form (64 bytes for $\mathbb{G}_1$, 128 bytes for $\mathbb{G}_2$) so that individual elements can be read without decompression overhead. This layout enables the chunked MSM to read base points sequentially from disk, matching the streaming model.

Remark 4.7 (Memory–group-operation trade-off)**.** The chunked Pippenger MSM offers a smooth trade-off between memory and group operations. At one extreme, $B_{\mathrm{MSM}} = N$ recovers the standard Pippenger bound ($\mathcal{O}(\lambda N/\log N)$ operations, $\Theta(N)$ memory). At the other extreme, $B_{\mathrm{MSM}} = \mathcal{O}(1)$ degrades to naïve scalar multiplication ($\mathcal{O}(\lambda N)$ operations, $\mathcal{O}(1)$ memory). The choice $B_{\mathrm{MSM}} = \Theta(\log N)$ in Sluice achieves a group-operation overhead of $\log N/\log\log N \approx 5.8$ at $N = 2^{28}$, which is modest compared to the memory savings from $\Theta(N)$ to $\Theta(\log N)$.

5. Sluice: The Full Construction

We assemble the streaming building blocks from Sections 3 and 4 into a complete Groth16 prover in the read-write streaming model. Setup and verification are identical to standard Groth16 [1]; only the prover is modified. The memory bound below is for the prove phase once the CRS, R1CS/QAP data, and assignment are available as streams; making setup itself streaming is an implementation task rather than a change to the proof system.

We recall the Groth16 prover's task. Given a QAP of degree d with $N = 2^{\lceil \log d \rceil}$, statement $\phi = (a_1, \ldots, a_\ell)$,

witness $w = (a_{\ell+1}, \ldots, a_m)$ ($a_0 = 1$), and CRS σ, the prover computes $\pi = ([A]_1, [B]_2, [C]_1)$ where

$$A = \alpha + \textstyle\sum_{i=0}^{m} a_i u_i(\tau) + r\delta, \tag{8}$$

$$B = \beta + \textstyle\sum_{i=0}^{m} a_i v_i(\tau) + s\delta, \tag{9}$$

$$C = \tfrac{1}{\delta}\big(\textstyle\sum_{i=\ell+1}^{m} a_i(\beta u_i(\tau) + \alpha v_i(\tau) + w_i(\tau)) + h(\tau)\,t(\tau)\big) + As + rB - rs\delta, \tag{10}$$

with random $r, s \leftarrow \mathbb{F}$ and quotient polynomial

$$h(X) = \frac{\big(\sum_i a_i\, u_i(X)\big)\big(\sum_i a_i\, v_i(X)\big) - \sum_i a_i\, w_i(X)}{t(X)}. \tag{11}$$

Our streaming prover replaces each stage with a read-write streaming subroutine, reducing the memory from $\mathcal{O}(N)$ to $\mathcal{O}(\log N)$. The key insight is that each step of the Groth16 prover—witness arrangement, R1CS evaluation, quotient polynomial computation, and MSM—can be reformulated as a sequence of streaming operations, where the intermediate results flow between stages as sequential streams on external storage. The composition lemma (Theorem 2.6) ensures that the overall memory remains $\mathcal{O}(\log N)$, since it is the maximum (not the sum) of the individual stages' memory requirements.

5.1. Streaming Witness Arrangement

The first step is to arrange the full assignment vector $(a_0, a_1, \ldots, a_m)$ from the statement ϕ and witness w. We follow the standard Groth16 convention that the prover receives the *complete* witness $w = (a_{\ell+1}, \ldots, a_m)$ as input, i.e., all wire values have already been computed by the prover prior to the proving phase.

Lemma 5.1 (Streaming witness arrangement)**.** *Given the statement $\phi = (a_1, \ldots, a_\ell)$ and the witness $w = (a_{\ell+1}, \ldots, a_m)$ as sequential streams, the full assignment $(a_0, a_1, \ldots, a_m)$ with $a_0 = 1$ can be arranged into a single output stream in the read-write streaming model using $\mathcal{O}(1)$ random-access memory, 1 pass, and $\mathcal{O}(m)$ streaming space.*

Proof. Write $a_0 = 1$ to the output stream, then copy the ℓ statement elements from the statement stream, then copy the $m - \ell$ witness elements from the witness stream. The only state is an $\mathcal{O}(1)$-element buffer for the current element being copied. □

5.2. Streaming R1CS Evaluation

Computing $h(X)$ via (11) requires the R1CS matrix-vector products $\hat{a}_q = \sum_i a_i \cdot U_{q,i}$ (and analogously for V, W), where $U_{q,i} = u_i(\omega^q)$ form the R1CS matrices padded to the N-point QAP domain by zero rows $d, \ldots, N-1$. We use Scribe's read-write streaming radix sort as a black-box primitive: S records with $\mathcal{O}(\log N)$-bit keys can be stably sorted by key using $\mathcal{O}(\log N)$ passes, $\mathcal{O}(\log N)$ RAM, $\mathcal{O}(S \log N)$ I/O, and $\mathcal{O}(S)$ external space. After sorting by row key, equal-key records are contiguous and can be reduced by a single scan.

Lemma 5.2 (Streaming sparse mat-vec)**.** *Let $M \in \mathbb{F}^{N \times (m+1)}$ be the matrix obtained from a d-row R1CS matrix by appending zero rows $d, \ldots, N-1$, with S nonzero entries, and let $\mathbf{a} \in \mathbb{F}^{m+1}$ be a vector given as a stream. The read-write streaming sparse mat-vec outputs the length-N stream $(M\mathbf{a})_0, \ldots, (M\mathbf{a})_{N-1}$ using $\mathcal{O}(\log N)$ random-access memory, $\mathcal{O}(S \log N)$ total I/O, and $\mathcal{O}(S)$ streaming space.*

Proof. Store M in column-sorted order (triples $(q, i, M_{q,i})$ sorted by i). Three phases: (1) *Multiply*—stream through the matrix and witness, emitting S pairs $(q,\, a_i \cdot M_{q,i})$; (2) *Sort*—radix-sort the pairs by q in $\lceil \log N \rceil$ streaming passes ($\mathcal{O}(S \log N)$ I/O, $\mathcal{O}(\log N)$ memory); (3) *Reduce*—scan the sorted stream, accumulating partial sums per row ($\mathcal{O}(1)$ memory) and emitting rows $0, \ldots, N-1$ in order. Rows with no contributions, including every padded row $d, \ldots, N-1$, output zero. The assignment stream is re-read for each of U, V, W. For fan-in-2 circuits ($S = \mathcal{O}(N)$), the total I/O is $\mathcal{O}(N \log N)$. □

5.3. Computing $h(X)$ via Streaming NTT

Given the R1CS evaluations from Section 5.2, we compute $h(X)$ ($\deg \le N-2$) via a coset-NTT pipeline. Let $\eta \in \mathbb{F}$ with $\eta^N \neq 1$, so $t(\eta\omega^j) = \eta^N - 1 \neq 0$ for all j.

Step 1. Coefficient recovery (3 iNTTs). Apply RW-NTT^{-1} (Remark 3.12) of size N to each of $\hat{\mathbf{a}}, \hat{\mathbf{b}}, \hat{\mathbf{c}}$, obtaining coefficient vectors of the aggregate QAP polynomials $F_U(X)$, $F_V(X)$, $F_W(X)$ (degree $\le N-1$).

Step 2. Coset shift + evaluation (3 passes + 3 NTTs). For each $f \in \{F_U, F_V, F_W\}$ with coefficients $(f_0, \ldots, f_{N-1})$: (a) *Coset shift* (1 pass, $\mathcal{O}(1)$ memory): compute $f'_i = f_i \cdot \eta^i$, maintaining η^i on-the-fly via a single register initialized to 1 and multiplied by η at each step. (b) Apply RW-NTT to $(f'_0, \ldots, f'_{N-1})$, yielding $f(\eta\omega^j)$ for $j = 0, \ldots, N-1$.

Step 3. Pointwise quotient (1 pass). For each $j = 0, \ldots, N-1$, compute

$$\hat{h}_j = \frac{F_U(\eta\omega^j) \cdot F_V(\eta\omega^j) - F_W(\eta\omega^j)}{\eta^N - 1}.$$

The denominator is a single precomputed constant. Three evaluation streams are read in lockstep; one output stream is written ($\mathcal{O}(1)$ memory).

Step 4. Inverse coset NTT (1 iNTT + 1 unshift pass). (a) Apply RW-NTT^{-1} of size N to $(\hat{h}_0, \ldots, \hat{h}_{N-1})$, producing $(c_0, \ldots, c_{N-1})$. (b) *Coset unshift* (1 pass, $\mathcal{O}(1)$ memory): recover $h_i = c_i \cdot \eta^{-i}$, maintaining η^{-i} on-the-fly. The output $(h_0, \ldots, h_{N-2})$ gives the coefficients of $h(X)$ ($c_{N-1} = 0$ since $\deg h \le N-2$).

Proposition 5.3 (Streaming $h(X)$ computation)**.** *The quotient polynomial $h(X)$ can be computed in the read-write streaming model using $\mathcal{O}(\log N)$ random-access memory, $\mathcal{O}(N \log N)$ total I/O, and $\mathcal{O}(\log N)$ passes.*

Proof. Seven RW-NTT calls at $\mathcal{O}(N \log N)$ I/O each (Theorem 3.10) plus five $\mathcal{O}(N)$-I/O streaming passes (coset shifts,

quotient, unshift). Memory is $\mathcal{O}(\log N)$ throughout; total I/O is $\mathcal{O}(N \log N)$. □

5.4. Proof Elements via Streaming MSM

With the witness and quotient streams available, we compute the three proof elements via streaming MSMs (Section 4.2). $[A]_1$ ((8)) requires one MSM of size $m+1$ over $\{[u_i(\tau)]_1\}$; $[B]_2$ ((9)) requires one $\mathbb{G}_2$-MSM (plus an auxiliary $\mathbb{G}_1$-MSM for $[B]_1$ needed in (10)); $[C]_1$ ((10)) decomposes into a witness MSM ($\sum_{i=\ell+1}^{m} a_i \cdot [(\beta u_i(\tau)+\alpha v_i(\tau)+w_i(\tau))/\delta]_1$) and a quotient MSM ($\sum_j h_j \cdot [\tau^j t(\tau)/\delta]_1$), plus scalar terms $s\cdot[A]_1 + r\cdot[B]_1 - rs\cdot[\delta]_1$.

Proposition 5.4 (Streaming proof elements)**.** *The three proof elements $[A]_1, [B]_2, [C]_1$ can be computed in the read-write streaming model using $\mathcal{O}(\log N)$ random-access memory, $\mathcal{O}(\lambda(m+N)/\log\log N)$ group operations (with MSM chunk size $B_{\mathrm{MSM}} = \Theta(\log N)$), and $\mathcal{O}(1)$ passes per MSM (at most 5 MSMs total).*

Proof. Direct application of Theorem 4.4: the five MSM streams have total length $\mathcal{O}(m+N)$, use $\mathcal{O}(\log N)$ memory, and each completes in one pass over the scalar and CRS streams. □

5.5. The Complete Sluice Prover

We now state the complete construction and its main correctness guarantee.

The complete prover protocol is given in Algorithm 2. Setup(R) is identical to Groth16 [1]; the CRS elements are additionally serialized as contiguous streams (one per MSM) and the R1CS matrices in column-sorted order. Stream materialization must bind stream lengths and digests to the CRS manifest, and each serialized group element is decoded with subgroup checks before use. Vfy is also identical to Groth16.

Algorithm 2: SluiceProve: Streaming Groth16 Prover

Input: R1CS R, DecodeAndCheck-validated CRS streams σ, statement $\phi = (a_1, \ldots, a_\ell)$, witness $w = (a_{\ell+1}, \ldots, a_m)$
Output: Proof $\pi = ([A]_1, [B]_2, [C]_1)$

Sample $r, s \leftarrow \mathbb{F}$

`/* Phase 1: Streaming witness arrangement (Theorem 5.1) */`
$\mathbf{a} \leftarrow (1 \| \phi \| w)$ as a sequential stream

`/* Phase 2: Streaming R1CS evaluation (Theorem 5.2) */`
for $M \in \{U, V, W\}$ **do**
  $\hat{\mathbf{m}} \leftarrow \mathsf{StreamMatVec}(M, \mathbf{a})$ `// multiply → sort → reduce`

`/* Phase 3: Quotient polynomial via streaming NTT (Theorem 5.3) */`
$(F_U(X), F_V(X), F_W(X)) \leftarrow$ $\mathsf{RW\text{-}NTT}^{-1}(\hat{\mathbf{a}})$, $\mathsf{RW\text{-}NTT}^{-1}(\hat{\mathbf{b}})$, $\mathsf{RW\text{-}NTT}^{-1}(\hat{\mathbf{c}})$
Coset-shift, RW-NTT, pointwise quotient, $\mathsf{RW\text{-}NTT}^{-1}$, unshift $\rightarrow$ $h(X)$

`/* Phase 4: Proof elements via streaming MSM (Theorem 5.4) */`
$[A]_1 \leftarrow \mathsf{StreamMSM}\big(\{a_i\}, \{[u_i(\tau)]_1\}\big) + [\alpha]_1 + r[\delta]_1$

$[B]_2 \leftarrow \mathsf{StreamMSM}\big(\{a_i\}, \{[v_i(\tau)]_2\}\big) + [\beta]_2 + s[\delta]_2$

$[B]_1 \leftarrow \mathsf{StreamMSM}\big(\{a_i\}, \{[v_i(\tau)]_1\}\big) + [\beta]_1 + s[\delta]_1$

$[C]_1 \leftarrow \mathsf{StreamMSM}\big(\{a_i\}_{i>\ell}, \{[\frac{\beta u_i + \alpha v_i + w_i}{\delta}]_1\}\big) +$ $\mathsf{StreamMSM}\big(\{h_j\}, \{[\frac{\tau^j t(\tau)}{\delta}]_1\}\big) + s\cdot[A]_1 + r\cdot [B]_1 - rs\cdot[\delta]_1$

return $\pi = ([A]_1, [B]_2, [C]_1)$

Theorem 5.5 (Sluice correctness)**.** *For every QAP instance R, statement ϕ, and valid witness w, the Sluice prover (Algorithm 2) outputs a proof π that is identical to the proof produced by the standard Groth16 prover (given the same randomness r, s).*

Proof. Each phase (witness arrangement, R1CS evaluation, quotient computation, MSM) produces identical intermediate values to the standard prover by Theorems 3.9, 4.5 and 5.2, including the auxiliary $[B]_1$ used only in the scalar term $r \cdot [B]_1$ of $[C]_1$. Hence the resulting proof $\pi = ([A]_1, [B]_2, [C]_1)$ is identical. See Section B for phase-by-phase verification. □

Theorem 5.6 (Sluice prover complexity)**.** *Let N be the padded QAP/NTT domain size, m the number of non-constant wires, S the total number of nonzero R1CS entries, $\lambda = |p|$ the scalar bit length, and let B_{MSM} be the MSM chunk-size parameter. Assume that the CRS, R1CS/QAP data, and witness are already materialized as private sequential streams in the layout consumed by Algorithm 2; setup, stream materialization, and cleanup are accounted for separately. The Sluice prover computes a Groth16 proof with:*

1) ***Random-access memory:*** $\mathcal{O}(\log N + B_{\mathrm{MSM}})$ *field/group elements.*
2) ***Streaming space:*** $\mathcal{O}(m+N+S)$ *(external storage for CRS, R1CS, and intermediate streams).*
3) ***Field operations:*** $\mathcal{O}(S \log N + N \log N)$.
4) ***Group operations:*** $\mathcal{O}(\lambda(m+N)/\log B_{\mathrm{MSM}})$.
5) ***Total I/O:*** $\mathcal{O}(m + S\log N + N \log N)$.
6) ***Passes:*** $\mathcal{O}(\log N)$.
7) ***Proof size:*** 3 *group elements (unchanged).*
8) ***Verification:*** 3 *pairings (unchanged).*

For bounded-fan-in circuits with no superlinear unused-wire padding ($S = \mathcal{O}(N)$ and $m = \mathcal{O}(N)$) and $B_{\mathrm{MSM}} = \Theta(\log N)$, this simplifies to $\mathcal{O}(\log N)$ RAM,

$\mathcal{O}(N \log N)$ *field/I/O work, and* $\mathcal{O}(\lambda N / \log\log N)$ *group operations.*

Proof. The memory bottleneck is $\mathcal{O}(\log N + B_{\mathrm{MSM}})$ (from the MSM chunks); the I/O bottleneck is $\mathcal{O}(m + S \log N + N \log N)$ (from witness/CRS scans, the R1CS radix sort, and 7 size-N NTT invocations). Proof size and verification are unchanged because the cryptographic setup and verification are identical to Groth16. The detailed per-phase accounting appears in Section B. □

Remark 5.7 (Proving time overhead)**.** For bounded-fan-in instances with $m = \mathcal{O}(N)$, the Sluice prover matches the standard prover's $\mathcal{O}(N \log N)$ field-operation count but incurs a $\mathcal{O}(\log N / \log\log N)$ group overhead from chunked MSM (Theorem 4.3). The dominant practical cost is the $\mathcal{O}(N \log N)$ I/O: the $h(X)$ pipeline performs 7 RW-NTTs ($\approx 28 \log N$ sequential passes), mitigated by high-bandwidth SSD throughput.

Remark 5.8 (Streaming setup)**.** The Setup algorithm writes the CRS elements as contiguous streams on external storage: one stream per MSM (containing the base points in the order required by the prover), and the R1CS matrices in column-sorted order (triples $(q, i, M_{q,i})$ sorted by column index i). This is a one-time cost amortized over all subsequent proof generations. The streaming layout adds no cryptographic overhead: it is merely a storage-format transformation of the standard CRS.

Remark 5.9 (Coset generator choice)**.** The coset generator $\eta \in \mathbb{F}$ in Section 5.3 must satisfy $\eta^N \neq 1$ so that $t(\eta\omega^j) = \eta^N - 1 \neq 0$ for all $j \in [N]$. A standard choice is $\eta = \omega_K$ where ω_K is a primitive K-th root of unity with $K > N$ and $K \nmid N$. For BN-254, a generator of the 2^{28}-th roots subgroup suffices for all $N \leq 2^{27}$. The coset shift and unshift passes (Section 5.3, Steps 2a and 4b) compute powers of η on-the-fly, using $\mathcal{O}(1)$ registers.

6. Security Analysis

Sluice is not a new proof system: it shares the same setup algorithm, the same verification algorithm, and the same proof structure as standard Groth16 [1]. The only modification is the *internal implementation* of the prover, which replaces random-access computation with read-write streaming subroutines. By Theorem 5.5, the Sluice prover produces a proof that is *identical* to the proof produced by the standard Groth16 prover on the same inputs and randomness. Consequently, the security properties of Groth16 carry over directly.

System and adversarial model. Throughout this section, we use the standard NIZK interface/security experiment: the adversary $\mathcal{A}$ receives the common reference string σ (or, in the simulation setting, a simulated CRS), and may interact with an honest prover only by observing its output π. *Storage leakage model.* External storage is part of the prover's private workspace, as in the read-write streaming model of Scribe [2]: the streams containing CRS elements, R1CS matrices, witness-derived values, NTT buffers, and partial products reside on local storage not accessible to $\mathcal{A}$. The adversary observes only (σ, ϕ, π). This is a precondition of the identity-reduction theorems, not a consequence of the cryptographic protocol. Setup, witness generation, swap, crash dumps, filesystem journals, snapshots, and recovery of deleted temporary files are outside the theorem's leakage model; the measured prototype targets trusted local storage, or controlled-cloud storage where the prover operator controls snapshots, swap, journaling, and data-retention policy. Verifier inputs and streamed CRS elements must be decoded using subgroup-checked encodings, and stream identity must be fixed by a length-and-digest manifest before proving. Unchecked deserialization, manifest mismatches, or malformed CRS streams are outside the theorem statements below.

6.1. Completeness

Theorem 6.1 (Completeness)**.** *If* (ϕ, w) *satisfies the QAP relation* R*, then the* Sluice *prover (Algorithm 2) outputs a proof* π *that is accepted by the Groth16 verifier with probability* 1 *(perfect completeness).*

Proof. By Theorem 5.5, the Sluice prover outputs the same proof $\pi = ([A]_1, [B]_2, [C]_1)$ as the standard Groth16 prover (given the same randomness). Since the verification algorithm is identical to standard Groth16, and the standard Groth16 scheme has perfect completeness [1, Theorem 1], the verifier accepts π. □

6.2. Knowledge Soundness

Theorem 6.2 (Knowledge soundness)**.** *In the storage leakage model above and in the generic group model,* Sluice *is knowledge-sound: for every efficient adversary* $\mathcal{A}$ *that produces an accepting proof* π *for a statement* ϕ*, there exists an efficient extractor* $\mathcal{X}$ *that, given the same randomness as* $\mathcal{A}$*, outputs a valid witness* w *such that* $(\phi, w) \in R$*, except with negligible probability.*

More precisely, the knowledge-soundness guarantee of Sluice *is* identical *to that of standard Groth16 [1, Theorem 2].*

Proof sketch. The reduction is the identity: any adversary $\mathcal{A}$ against Sluice is also an adversary against standard Groth16, because the CRS distribution, the verification equation, and the proof structure are identical. The extractor from [1, Theorem 2] therefore applies without modification. No rewinding or simulation is needed, and the generic-group-model assumptions are not strengthened. See Section C for the full proof. □

6.3. Zero-Knowledge

Theorem 6.3 (Perfect zero-knowledge)**.** *In the storage leakage model above,* Sluice *is* perfect zero-knowledge*: there exists a simulator* Sim *that, given the relation* R*, the trapdoor*

τ_{trap}, *and the statement* ϕ *(but no witness), produces proofs whose distribution is identical to that of honestly generated proofs.*

The zero-knowledge guarantee is the same as that of standard Groth16 [1, Theorem 3].

Proof sketch. The simulator is the standard Groth16 simulator [1, Theorem 3], which depends only on the trapdoor and the statement. Because it never invokes the prover, the streaming modification is invisible. By Theorem 5.5, the real proof distributions of Sluice and standard Groth16 coincide, so the indistinguishability argument applies verbatim. See Section C for the full proof. □

6.4. Structural Argument

We emphasize the structural reason why the security proof is straightforward. A SNARK's security properties—soundness and zero-knowledge—are defined with respect to the *external* interface of the system: the CRS distribution, the proof distribution, and the verification predicate. Sluice modifies none of these:

- **CRS:** Generated by the standard Groth16 setup (Section 2.4), then reformatted into streaming-friendly layout (Theorem 4.6). The reformatting is a deterministic bijection that does not alter the mathematical content of the CRS elements.
- **Proof distribution:** By Theorem 5.5, for any fixed randomness (r, s) the Sluice proof is *bit-for-bit identical* to the standard Groth16 proof. Since the randomness distribution is unchanged (uniform over $\mathbb{F}^2$), the proof distributions coincide.
- **Verification:** The verifier is unchanged; it checks the same pairing equation (4).

Because the external interface is identical, any reduction to the hardness of the generic group model [27] that works for Groth16 works verbatim for Sluice. Thus the proof-system security loss is zero. This statement does not claim confidentiality if the prover's private storage is compromised.

Malleability boundary. Sluice preserves Groth16's exact security profile, including its usual application-level malleability considerations. It neither improves nor worsens these properties; deployments requiring simulation extractability or non-malleability must use standard Groth16-level transformations or external binding mechanisms.

Discussion. The identity-reduction argument above relies on the fact that Sluice modifies only the prover's *internal* computation strategy, not the cryptographic protocol itself. This is a strictly weaker requirement than designing a new proof system: no new assumptions are needed, no new extraction or simulation techniques are required, and the security reduction has no loss because it is the identity reduction. We emphasize that this property holds because Theorem 5.5 establishes *identical* proof distributions, not merely statistically close ones—a stronger guarantee than typically achievable when modifying a proof system.

Operational storage risks. The current prototype stores `FileVec` streams as plaintext temporary files. These files include witness values, QAP evaluations, quotient coefficients, NTT buffers, and MSM partial products; recovering them can reveal witness-derived information even though the final Groth16 proof remains zero-knowledge. Implementations that place such streams on untrusted or recoverable media should add authenticated stream encryption under a fresh per-proof session key, with unique per-stream nonces and best-effort unlinking after proof generation. Cloud-provider administrators and platform snapshots are adversarial unless controlled by the prover operator or protected by this stream layer. We do not rely on secure overwrite semantics: on modern SSDs, wear levelling makes reliable physical deletion difficult to guarantee. This hardening changes only the storage layer, not the Groth16 proof distribution; Section D reports storage-layer and small end-to-end AEAD measurements, while production key management remains outside this prototype.

7. Complexity Analysis and Comparison

We now summarize the end-to-end complexity of Sluice and compare it with the standard Groth16 prover [1] and the Scribe streaming SNARK [2], which applies read-write streaming to HyperPlonk [3].

7.1. Complexity Summary

Theorem 5.6 establishes the complexity of the Sluice prover in full generality, including the wire-count parameter m. For the common case of bounded-fan-in circuits with no superlinear unused-wire padding ($S = \mathcal{O}(N)$ and $m = \mathcal{O}(N)$) and MSM chunk size $B_{\text{MSM}} = \Theta(\log N)$, the key figures are:

Metric	Sluice
RAM	$\mathcal{O}(\log N)$ field/group elements
Ext. storage	$\mathcal{O}(N)$
Field ops	$\mathcal{O}(N \log N)$
Group ops	$\mathcal{O}(\lambda N / \log\log N)$
Total I/O	$\mathcal{O}(N \log N)$
Passes	$\mathcal{O}(\log N)$
Proof size	3 group elements
Verification	3 pairings + $\mathcal{O}(\ell)$ $\mathbb{G}_1$ ops

Here N is the padded QAP/NTT domain size, m is the nonconstant wire count, $\lambda = |p|$ is the scalar bit length, and ℓ is the number of public inputs.

7.2. Comparison with Prior Work

Table 2 compares Sluice with the standard Groth16 prover and with Scribe (applied to HyperPlonk).

Key observations. Sluice and standard Groth16 share the same verifier-facing interface: 3 proof elements, 3 pairings, and $\mathcal{O}(\ell)$ public-input group operations. Scribe preserves the verifier profile of its

	Groth16	Sluice	Scribe
Proof size	$\mathcal{O}(1)$	$\mathcal{O}(1)$	PC-dependent
Verification	$\mathcal{O}(\ell)$	$\mathcal{O}(\ell)$	PC-dependent
Field ops	$\mathcal{O}(N \log N)$	$\mathcal{O}(N \log N)$	$\mathcal{O}(N \log N)$
Group ops	$\mathcal{O}\left(\frac{\lambda N}{\log N}\right)$	$\mathcal{O}\left(\frac{\lambda N}{\log \log N}\right)$	PC ops
Prover RAM	$\mathcal{O}(N)$	$\mathcal{O}(\log N)$	$\mathcal{O}(\log N)$
Ext. storage	—	$\mathcal{O}(N)$	$\mathcal{O}(N)$
Passes	1	$\mathcal{O}(\log N)$	$\mathcal{O}(1)$

TABLE 2. ASYMPTOTIC COMPARISON OF SNARK PROVERS. N: PADDED QAP/NTT DOMAIN SIZE; λ: SCALAR BIT LENGTH; ℓ: PUBLIC INPUTS. THE RAM ROW MEANS RANDOM-ACCESS WORKING MEMORY; ALL PROVERS STILL CONSUME $\mathcal{O}(N)$ TOTAL CRS/INPUT OR EXTERNAL-STORAGE FOOTPRINT. SCRIBE'S PROVER GROUP/COMMITMENT WORK IS NOT DIRECTLY COMPARABLE TO GROTH16 MSMS BECAUSE IT USES THE HYPERPLONK POLYNOMIAL-COMMITMENT PIPELINE RATHER THAN GROTH16 CRS MSMS.

underlying HyperPlonk/polynomial-commitment instantiation, so its concrete proof elements and verifier work are not scheme-independent. Both Sluice and Scribe achieve $\mathcal{O}(\log N)$ RAM; the difference is that SBM (Section 3) handles Groth16's non-unit butterfly strides. All three provers have $\mathcal{O}(N \log N)$ field operations in this bounded-fan-in specialization; Sluice's Groth16-MSM group cost is $\mathcal{O}(\lambda N/\log \log N)$ when $m = \mathcal{O}(N)$ (a $\log N/\log \log N \approx 5.8\times$ overhead at $N = 2^{28}$). Sluice requires $\mathcal{O}(\log N)$ passes (vs. Scribe's $\mathcal{O}(1)$); concretely, the $h(X)$ pipeline performs 7 RW-NTT invocations, each with at most $4 \log N + 1$ streaming scans.

7.3. Trade-off Analysis

Trade-offs. Sluice produces proofs *byte-for-byte identical* to standard Groth16 (given the same randomness), so it is a verifier-compatible proving path requiring no verifier changes. Scribe's $\mathcal{O}(1)$-pass I/O and NTT-free prover are preferable when verifier compatibility is not required.

Verifier-contract compatibility. Groth16 verification matters not just because it uses a constant number of pairings, but because the interface is already deployed. On BN-254, a Groth16 verifier uses the standard pairing precompile for a fixed 3-pairing equation plus $\mathcal{O}(\ell)$ public-input group operations. Sluice preserves this exact verifier and proof format. In contrast, replacing Groth16 by a HyperPlonk/Scribe-style prover requires adopting the verifier of the chosen polynomial-commitment instantiation; its gas cost and proof elements are scheme-specific and not directly comparable to Groth16 CRS MSMs.

Concrete overhead estimates. For the largest same-size standard-vs-Full-RW comparison in our evaluation, $N = 2^{23}$ (≈ 8.4 million padded QAP rows) on BN-254 ($\lambda = 254$):

- The standard Groth16 prover's NTT requires $N = 2^{23}$ field elements in RAM, which at 32 bytes per element is ≈ 256 MiB for one NTT array alone; including the prover's other arrays and CRS, our isolated standard run reaches 3 418 MB prove-phase ΔRSS.
- The Sluice prover uses $\mathcal{O}(\log N) = 23$ field-element registers for the NTT and $2^w = 256$ group-element buckets for the MSM (with $w = 8$), totalling ≈ 16 KB of algorithmic live state. Process-level RSS is larger because it includes I/O buffers, runtime/library state, allocator overhead, and OS accounting.
- The asymptotically tuned MSM schedule of Theorem 4.4 has overhead $\log N/\log \log N = 23/\log 23 \approx 5.1\times$ relative to standard Pippenger. The prototype instead fixes $w = 8$ and $B_{\mathrm{MSM}} = 2^w = 256$, giving an implemented MSM cost $\mathcal{O}(\lambda N/8)$ and a concrete window-overhead factor $\approx 23/8 = 2.9\times$ at $N = 2^{23}$.
- One SBM NTT performs at most $4 \log N + 1 \approx 93$ sequential scans. The Groth16 quotient pipeline invokes 7 such transforms, for roughly $28 \log N$ scans plus linear coset and quotient passes. This is the concrete source of the $\mathcal{O}(\log N)$ pass overhead.

Remark 7.1 (Memory–time trade-off)**.** Increasing the buffer from $\mathcal{O}(\log N)$ to $\mathcal{O}(N^\epsilon)$ reduces SBM passes to $\mathcal{O}((1-\epsilon) \log N)$ and MSM overhead to $\mathcal{O}(\lambda N/(\epsilon \log N))$, approaching the standard Pippenger bound at $\epsilon = 1$. Intermediate values of ϵ (e.g., $\epsilon = 1/2$, giving $\mathcal{O}\left(\sqrt{N}\right)$ memory) may be attractive for servers with moderate RAM.

Comparison with Hobbit. Hobbit [19] also targets space-efficient SNARK proving but takes a fundamentally different approach: it redesigns the underlying polynomial IOP to achieve space efficiency, resulting in a new proof system. In contrast, Sluice retrofits the existing Groth16 construction via streaming, preserving the exact same CRS, proof format, and verification equation. Thus Sluice is a verifier-compatible option for systems that already use a Groth16 verifier, including deployed smart contracts on Ethereum, while changing the prover's storage and I/O profile. Hobbit's approach offers the advantage of avoiding the $\mathcal{O}(\log N)$ I/O overhead inherent in SBM, but requires a new verifier implementation and cannot leverage the existing Groth16 ecosystem.

8. Implementation and Evaluation

We implement Sluice in Rust using `ark-bn254` v0.5 for BN-254 arithmetic; the prototype contains about 3 750 Rust lines and 42 passing unit tests. The anonymized implementation artifact is available at https://anonymous.4open.science/r/rw-groth16-artifact/; Section D maps the reported results to raw CSVs, logs, and reproduction commands.

8.1. Implementation Details

FileVec abstraction. The implementation centers on `FileVec<T>`, a disk-backed sequential array with 32-byte field-element serialization, 256 KB read/write buffers, and RAII cleanup. Reported large-scale runs use plaintext

streams on trusted local storage; untrusted media require authenticated encryption. The optional AEAD `FileVec` mode is evaluated in Section D. All SBM intermediates are `FileVec` streams, so the transform size does not create RAM-resident arrays.

Streaming CRS and MSM. To realize the $\mathcal{O}(\log N)$ RAM bound of Theorem 4.4, the five MSM CRS arrays live in a `StreamingProvingKey` backed by `FileVec<G1Disk>` and `FileVec<G2Disk>`. Elements are uncompressed (64 bytes for $\mathbb{G}_1$, 128 bytes for $\mathbb{G}_2$), and `file_chunk_msm` scans them in chunks of $B_{\mathrm{MSM}} = 2^w$ base/scalar pairs. The main RAM allocation per MSM is the 2^w bucket array, 16 384 bytes for $w = 8$. Stream deserialization must reject malformed or non-subgroup group encodings; deployments should also authenticate CRS stream lengths and digests before proving. Unchecked or swapped CRS streams are outside the implementation and security claims. For bounded-fan-in circuits with $m \approx N$ wires, this uncompressed streaming CRS stores roughly $4N$ $\mathbb{G}_1$ elements and N $\mathbb{G}_2$ elements, or about $384N$ bytes: ≈ 3.0 GiB at $N = 2^{23}$. This is persistent storage, not random-access working memory, and contains the same algebraic CRS material as the standard proving key in a sequential layout.

Fully streaming R1CS and witness. A previous implementation (streaming_prove_disk) accepted the assignment as a Rust slice `&[Fr]` in the caller's heap, reintroducing an $\mathcal{O}(N)$ RAM cost in the prove path. The current implementation (streaming_prove_rw) stores the assignment, QAP matrices, and CRS as `FileVec` streams; its R1CS routine reads matrix entries and assignment values from disk and writes the output polynomial to disk. No $\mathcal{O}(N)$ `Vec` allocation appears in the prove path.

On-the-fly twiddle factors. Twiddle factors are computed incrementally (Theorem 3.4) from one running register and one step constant, avoiding precomputed lookup tables.

Experimental setup. Measurements use an Apple M1 Pro MacBook Pro with 32 GB memory, macOS Tahoe 26.5, BN-254, and `--release` binaries. Std is our Rust Groth16 prover on the same instances; arkworks denotes an arkworks reference baseline, not a same-methodology clean prove-only memory baseline. Full-RW denotes the final prove-only streaming worker: after CRS, QAP, and witness streams are materialized, a fresh process reopens them through sequential `FileVec` streams and computes only the proof, avoiding setup's monotone `ru_maxrss` high-water mark. Direct scaling reports three-run means at $N = 2^{18}, 2^{20}, 2^{22}$ and single runs at $N = 2^{23}, 2^{24}, 2^{25}$. Bounded-memory workers run after the same materialization step in a Docker Linux backend (`aarch64`, cgroup v2, `MemTotal=25425485824`) with `--memory=cap`, `--memory-swap=cap`, and `/usr/bin/time -v`; those logs are used only for cap success/failure, not macOS RSS.

Measurement boundary. The $\mathcal{O}(\log N)$ result is a prove-phase working-memory claim after CRS/QAP/witness stream materialization, not an end-to-end setup or storage-footprint claim. Setup may allocate $\mathcal{O}(N)$ transient state; storage is trusted by default, with AEAD outside the theorem.

Worker	$\log_2 N$	Cap	Outcome
Full-RW	23	8 GB	success; 45.3 min; 29.1 MB; valid 128 B
Std	23	8 GB	SIGKILL; 21.6 s; 7.97 GiB; no proof
Std	23	12 GB	SIGKILL; 32.4 s; 11.95 GiB; no proof
Std	23	16 GB	success; 10.9 min; 13.3 GiB; valid 128 B
Full-RW	25	8 GB	success; 181.6 min; 29.4 MB; valid 128 B

TABLE 3. BOUNDED-MEMORY PROVE-ONLY WORKERS AFTER STREAM MATERIALIZATION. THE SAME-SIZE STANDARD-VS-FULL-RW COMPARISON IS AT $N = 2^{23}$; THE $N = 2^{25}$ ROW IS FULL-RW-ONLY.

8.2. SBM NTT Microbenchmark

In Appendix D, Figure 2(a) compares the RW-NTT (using `FileVec` streams) against the in-memory `ark-fft` routine from `ark-poly` for $\log_2 N$ from 8 to 16. Overhead ranges from $1.9\times$ at $N = 2^8$ to $5.9\times$ at $N = 2^{16}$, consistent with SBM's multi-scan design; all outputs match `ark-fft` exactly.

8.3. End-to-End Scalability

The standard-vs-Full-RW bounded-memory comparison is centered on the $N = 2^{23}$ cgroup experiment in Table 3. Direct Full-RW scaling through $N = 2^{25}$ is reported in Table 4 and Figure 1 of Appendix D. The $N = 2^{25}$ point is a timing/I/O result plus a Full-RW-only 8 GB capped success check.

Interpretation. Table 3 gives the largest same-size comparison: Full-RW is $2.5\times$ slower at $N = 2^{23}$ but lowers macOS clean-worker peak RSS by $383\times$ while preserving the 128-byte proof and standard verifier. This peak-RSS ratio is distinct from the 3 418 MB prove-phase ΔRSS estimate in Section 7.3 and the Linux cgroup MaxRSS in the table. The cgroup block gives the memory boundary: Std is killed at 8 GB and 12 GB and succeeds at 16 GB, whereas pre-materialized Full-RW succeeds at 8 GB. The $N = 2^{25}$ capped row is Full-RW-only; direct scaling/I/O and structured-workload details appear in Tables 4 and 5 and Figure 1 of Appendix D. `FileVec` counters report 432.6/1 872.3 GiB of application payload at $N = 2^{23}/2^{25}$, giving 285.6/302.2 MiB/s effective sequential throughput; metadata and page-cache traffic are excluded.

9. Conclusion

We presented Sluice, a read-write streaming Groth16 prover with $\mathcal{O}(\log N)$ prove-phase RAM, 128-byte proofs, and the standard verifier. At $N = 2^{23}$, it cuts macOS clean-worker peak RSS by $383\times$ and succeeds under an 8 GB cap, while Std is killed at 8 GB and 12 GB and succeeds at 16 GB. The scope is pre-materialized private streams; production-key AEAD, streaming setup, and the $\mathcal{O}(N)$ CRS/storage footprint remain future work.

References

[1] J. Groth, "On the size of pairing-based non-interactive arguments," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2016, pp. 305–326.

[2] A. Baweja, P. Mishra, T. Mopuri, K. Newatia, and S. Wang, "Scribe: Low-memory SNARKs via read-write streaming," in *35th USENIX Security Symposium (USENIX Security 26)*, 2026, ePrint 2024/1970.

[3] B. Chen, B. Bünz, D. Boneh, and Z. Zhang, "Hyperplonk: Plonk with linear-time prover and high-degree custom gates," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2023, pp. 499–530.

[4] D. H. Bailey, "Ffts in external or hierarchical memory," *The journal of Supercomputing*, vol. 4, no. 1, pp. 23–35, 1990.

[5] H. Jia-Wei and H.-T. Kung, "I/o complexity: The red-blue pebble game," in *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, 1981, pp. 326–333.

[6] A. Aggarwal and S. Vitter, Jeffrey, "The input/output complexity of sorting and related problems," *Communications of the ACM*, vol. 31, no. 9, pp. 1116–1127, 1988.

[7] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *40th Annual Symposium on Foundations of Computer Science (FOCS '99)*. IEEE, 1999, pp. 285–297.

[8] R. Gennaro, C. Gentry, B. Parno, and M. Raykova, "Quadratic span programs and succinct nizks without pcps," in *Annual international conference on the theory and applications of cryptographic techniques*. Springer, 2013, pp. 626–645.

[9] B. Parno, J. Howell, C. Gentry, and M. Raykova, "Pinocchio: Nearly practical verifiable computation," *Communications of the ACM*, vol. 59, no. 2, pp. 103–112, 2016.

[10] E. Ben-Sasson, A. Chiesa, E. Tromer, and M. Virza, "Succinct {Non-Interactive} zero knowledge for a von neumann architecture," in *23rd USENIX Security Symposium (USENIX Security 14)*, 2014, pp. 781–796.

[11] J. Groth and M. Maller, "Snarky signatures: Minimal signatures of knowledge from simulation-extractable snarks," in *Annual International Cryptology Conference*. Springer, 2017, pp. 581–612.

[12] S. Bowe, A. Gabizon, and I. Miers, "Scalable multi-party computation for zk-snark parameters in the random beacon model," *Cryptology ePrint Archive*, 2017.

[13] C. Lund, L. Fortnow, H. Karloff, and N. Nisan, "Algebraic methods for interactive proof systems," *Journal of the ACM (JACM)*, vol. 39, no. 4, pp. 859–868, 1992.

[14] S. Goldwasser, Y. T. Kalai, and G. N. Rothblum, "Delegating computation: interactive proofs for muggles," *Journal of the ACM (JACM)*, vol. 62, no. 4, pp. 1–64, 2015.

[15] S. Setty, "Spartan: Efficient and general-purpose zksnarks without trusted setup," in *Annual International Cryptology Conference*. Springer, 2020, pp. 704–737.

[16] A. Gabizon, Z. J. Williamson, and O. Ciobotaru, "Plonk: Permutations over lagrange-bases for oecumenical noninteractive arguments of knowledge," *Cryptology ePrint Archive*, 2019.

[17] A. Chiesa, Y. Hu, M. Maller, P. Mishra, N. Vesely, and N. Ward, "Marlin: Preprocessing zksnarks with universal and updatable srs," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2020, pp. 738–768.

[18] E. Ben-Sasson, I. Bentov, Y. Horesh, and M. Riabzev, "Scalable, transparent, and post-quantum secure computational integrity," *Cryptology ePrint Archive*, 2018.

[19] C. Pappas and D. Papadopoulos, "Hobbit: Space-efficient zkSNARK with optimal prover time," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 4917–4936.

[20] E. B. Sasson, A. Chiesa, C. Garman, M. Green, I. Miers, E. Tromer, and M. Virza, "Zerocash: Decentralized anonymous payments from bitcoin," in *2014 IEEE symposium on security and privacy*. IEEE, 2014, pp. 459–474.

[21] V. Buterin, "An incomplete guide to rollups," https://vitalik.ca/general/2021/01/05/rollup.html, 2021.

[22] Zcash Foundation, "bellman: A zk-SNARK library," 2019, https://github.com/zkcrypto/bellman.

[23] arkworks contributors, "arkworks: An ecosystem for developing and programming with zkSNARKs," 2022, https://arkworks.rs.

[24] iden3, "rapidsnark: Fast Groth16 prover," 2021, https://github.com/iden3/rapidsnark.

[25] J. W. Cooley and J. W. Tukey, "An algorithm for the machine calculation of complex fourier series," *Mathematics of computation*, vol. 19, no. 90, pp. 297–301, 1965.

[26] N. Pippenger, "On the evaluation of powers and monomials," *SIAM Journal on Computing*, vol. 9, no. 2, pp. 230–250, 1980.

[27] V. Shoup, "Lower bounds for discrete logarithms and related problems," in *EUROCRYPT*. Springer, 1997, pp. 256–266.

Appendix A. Deferred Proofs for the Streaming NTT

Proof of 3.2 (Pairing lemma). Write p in binary as $p = (b_{n-1}, \ldots, b_{j+1}, 0, b_{j-1}, \ldots, b_0)$. Then

$$p \oplus 2^j = (b_{n-1}, \ldots, b_{j+1}, 1, b_{j-1}, \ldots, b_0) \in \mathcal{B}_j.$$

The rank $\sigma_j(p)$ equals the integer whose binary representation is

$$(b_{n-1}, \ldots, b_{j+1}, b_{j-1}, \ldots, b_0),$$

the $(n-1)$-bit string obtained by deleting bit j. The same $(n-1)$-bit string determines the rank $\tau_j(p \oplus 2^j)$. Hence the two ranks coincide. □

Proof of Theorem 3.3 (Twiddle factor identity). The lower j bits of p are $(b_{j-1}, \ldots, b_0)$, which form the lower j bits of the rank $q = \sigma_j(p)$ since σ_j merely removes bit j and shifts higher bits down by one position. □

Proof of Theorem 3.4 (On-the-fly twiddle). Let $r = q \bmod 2^j$. Since ω_{cur} resets to 1 at every index q that is a multiple of 2^j, the value after r subsequent multiplications is

$$\omega_{\text{cur}}^{(q)} = W_j^{\,r} = \left(\omega^{2^{n-j-1}}\right)^r = \omega^{r \cdot 2^{n-j-1}}.$$

By 3.3, $r = q \bmod 2^j = p \bmod 2^j$ where $p = \sigma_j^{-1}(q) \in \mathcal{A}_j$, so the expression equals ω_q.

The counter tracks r via $r \leftarrow r + 1$, resetting to 0 when $r = 2^j$. This requires $j \le n$ bits, i.e., $\mathcal{O}(\log N)$ bits. No other state is needed. □

Proof of Theorem 3.5 (SBM correctness). Let $x_0, \ldots, x_{N-1}$ be the input in natural order and let $y_0, \ldots, y_{N-1}$ be the output of the standard stage-j computation. For each $p \in \mathcal{A}_j$:

$$y_p = x_p + \omega^{(p \bmod 2^j) \cdot 2^{n-j-1}} \cdot x_{p \oplus 2^j},$$
$$y_{p \oplus 2^j} = x_p - \omega^{(p \bmod 2^j) \cdot 2^{n-j-1}} \cdot x_{p \oplus 2^j}.$$

After Split, stream $\mathcal{S}_A$ contains $\{x_p : p \in \mathcal{A}_j\}$ in order, and $\mathcal{S}_B$ contains $\{x_{p'} : p' \in \mathcal{B}_j\}$ in order. By Theorem 3.2, the q-th elements satisfy $a_q = x_p$ and $b_q = x_{p \oplus 2^j}$ where $q = \sigma_j(p)$. By Theorem 3.3, the twiddle ω_q at index q equals $\omega^{(p \bmod 2^j) \cdot 2^{n-j-1}}$. Thus PairedBfly produces $y_q^{(\mathrm{top})} = y_p$ in O_A and $y_q^{(\mathrm{bot})} = y_{p \oplus 2^j}$ in O_B.

It remains to show that Merge reads each of O_A and O_B sequentially while producing the output in natural order. The output positions $p = 0, 1, \ldots, N-1$ are visited in order. Among these, the positions with $\mathsf{bit}_j(p) = 0$ form contiguous blocks of 2^j (namely $[k \cdot 2^{j+1},\ k \cdot 2^{j+1} + 2^j - 1]$ for $k = 0, 1, \ldots$), alternating with blocks where $\mathsf{bit}_j(p) = 1$. Within each block of 2^j positions reading from O_A, the corresponding indices in O_A are consecutive: the positions p in the k-th such block have ranks $\sigma_j(p) = k \cdot 2^j, \ldots, k \cdot 2^j + 2^j - 1$ (since σ_j is order-preserving and bit j is the only distinguishing bit removed). The same argument applies to O_B via τ_j. Hence both streams are read front-to-back without seeking, and the q-th element of O_A (which is y_p with $\sigma_j(p) = q$) is placed at the correct output position p. This recovers $y_0, \ldots, y_{N-1}$ in natural order. □

Proof of Theorem 3.7 (Bit-swap correctness). Consider an element x_p originally at position p. After the split, x_p resides in stream $S_{b_i b_j}$ where $b_i = \mathsf{bit}_i(p)$ and $b_j = \mathsf{bit}_j(p)$. Its target position is p' with $\mathsf{bit}_i(p') = b_j$ and $\mathsf{bit}_j(p') = b_i$ (all other bits unchanged). During the cross-merge, position p' has $(\mathsf{bit}_i(p'), \mathsf{bit}_j(p')) = (b_j, b_i)$, and the rule reads from stream $S_{b_i b_j}$. Since elements within each $S_{b_i b_j}$ preserve their relative order, and the merge visits output positions sequentially, the element x_p appears at position p'.

Memory consists of a position counter ($\mathcal{O}(\log N)$ bits) and the four stream pointers, totalling $\mathcal{O}(\log N)$. □

Proof of Theorem 3.8 (Streaming bit-reversal). We compose $\lfloor n/2 \rfloor$ bit-swaps:

$$\mathsf{BitSwap}(\cdot, 0, n-1) \circ \mathsf{BitSwap}(\cdot, 1, n-2) \circ \cdots .$$

Each BitSwap uses 2 passes; the intermediate streams of one swap are recycled for the next. Total passes: $2\lfloor n/2 \rfloor \le n$. Memory and stream counts are constant per swap and do not accumulate by the *composition lemma* of [2]: when two streaming algorithms are composed sequentially (the output stream of one becoming the input stream of the next), the intermediate streams can be reused, so the total number of simultaneously active streams and the random-access memory are bounded by the maximum over the individual algorithms, not their sum. □

Proof of Theorem 3.9 (RW-NTT correctness). The standard DIT NTT consists of a bit-reversal permutation followed by n butterfly stages. The bit-reversal is handled correctly by Theorem 3.8. For each stage j:

- If $2^{j+1} \le c \log N$, the entire butterfly group fits in memory, and DirectBfly computes the standard butterflies exactly.
- Otherwise, SBM_j produces the correct stage-j output by Theorem 3.5.

Since each stage takes as input the output of the previous stage (in natural order) and produces its output in natural order, the composition of all stages yields the correct NTT result. □

Proof of Theorem 3.10 (RW-NTT complexity). We account for each component:

Bit-reversal. By Theorem 3.8: at most n passes, each reading and writing N elements. I/O: $\mathcal{O}(N \cdot n) = \mathcal{O}(N \log N)$. Memory: $\mathcal{O}(\log N)$.

Small stages (j such that $2^{j+1} \le c \log N$, *i.e.* $j \le \log\log N + \mathcal{O}(1)$). Each stage uses 1 pass and $\mathcal{O}(\log N)$ memory. Total passes: $\mathcal{O}(\log\log N)$.

Large stages ($j > \log\log N + \mathcal{O}(1)$). Each stage uses 3 passes (Split, PairedBfly, Merge), each reading and writing N elements. The number of large stages is at most $n - \log\log N = \mathcal{O}(n)$. Total passes: $\mathcal{O}(3n)$.

Totals. Passes: $n + \mathcal{O}(\log\log N) + 3(n - \log\log N) \le 4n + \mathcal{O}(1)$. I/O per pass: $\mathcal{O}(N)$. Total I/O: $\mathcal{O}(N \cdot 4n) = \mathcal{O}(N \log N)$.

Memory never exceeds $\mathcal{O}(\log N)$: the bit-reversal uses $\mathcal{O}(\log N)$, DirectBfly buffers at most $c \log N$ elements, and the SBM subroutines use a position counter plus constant field-element registers. Intermediate streams are reused across stages by the composition lemma of [2], so at most 4 streams of size N are active at any time.

The butterfly arithmetic is the same as in the standard radix-2 NTT: $\frac{N}{2} \log N$ butterflies, each requiring one multiplication and two additions. On-the-fly twiddle maintenance adds only a constant number of field operations per butterfly, so the asymptotic field-operation count remains $\mathcal{O}(N \log N)$. □

Appendix B. Deferred Proofs for the Full Construction

Details for Theorem 5.2. The matrix stream is a sequence of triples $(q, i, M_{q,i})$ sorted by column index i; ties are ordered arbitrarily but deterministically. The assignment stream stores (i, a_i) in increasing i. During the multiply phase, the prover advances the assignment stream until the current column matches i, emits $(q, a_i M_{q,i})$, and reuses that assignment value for all consecutive triples with the same i. This is a sequential merge join and needs only the current triple, current assignment entry, and counters. The emitted pairs are sorted by row index q using the read-write streaming radix sort primitive of Scribe [2]; stability is not required because equal-key values are summed by the next phase. The reducer scans the sorted stream, accumulates all values with the same q, emits zeros for missing row indices, and outputs rows in order $0, \ldots, N-1$. Since rows $d, \ldots, N-1$ are padded zero constraints, they have no input triples and are emitted as zero. Thus the stream entering each size-N inverse NTT is exactly the standard padded QAP evaluation vector. For U, V, W the assignment stream is reopened from the beginning; this adds a constant factor and does not change the asymptotic I/O bound.

Proof of 5.5 (Sluice correctness). We verify that each phase computes the same values as the standard prover:

- *Witness*: the full assignment $(a_0, \ldots, a_m)$ is assembled by concatenating $a_0 = 1$, the statement, and the witness—identical to the standard prover's input.
- *R1CS evaluation*: the sparse mat-vec is a deterministic linear-algebraic operation over the padded N-row QAP domain; the streaming algorithm computes the length-N vectors exactly, including zero padded rows (Theorem 5.2).
- *Quotient polynomial*: each RW-NTT invocation produces the same output as the standard NTT (Theorem 3.9), and the pointwise operations are identical. Thus $h(X)$ has the same coefficients.
- *Proof elements*: the streaming MSM computes the same linear combination as the standard MSM (Theorem 4.5), so A, B, C are identical.

Since $\pi = ([A]_1, [B]_2, [C]_1)$ is the same, the standard Groth16 verifier accepts. □

Proof of Theorem 5.6 (Sluice prover complexity). We sum the costs of each phase:

- *Witness arrangement:* $\mathcal{O}(1)$ memory, $\mathcal{O}(m)$ I/O, 1 pass.
- *R1CS evaluation:* $\mathcal{O}(\log N)$ memory, $\mathcal{O}(S \log N)$ I/O (Theorem 5.2, applied three times for U, V, W).
- $h(X)$ *computation:* $\mathcal{O}(\log N)$ memory, $\mathcal{O}(N \log N)$ field ops and I/O, $\mathcal{O}(\log N)$ passes (Theorem 5.3).
- *Proof MSMs:* $\mathcal{O}(B_{\mathrm{MSM}})$ memory, $\mathcal{O}(\lambda(m+N)/\log B_{\mathrm{MSM}})$ group ops, 5 passes (Theorem 5.4).

The memory bottleneck is $\mathcal{O}(\log N + B_{\mathrm{MSM}})$; with $B_{\mathrm{MSM}} = \Theta(\log N)$ this is $\mathcal{O}(\log N)$. The I/O bottleneck is $\mathcal{O}(m + S \log N + N \log N)$ (from witness/CRS scans, the R1CS radix sort, and the 7 size-N NTT invocations in the $h(X)$ pipeline). The total number of passes is dominated by the NTT phases: $7 \cdot \mathcal{O}(\log N) = \mathcal{O}(\log N)$.

The proof size and verification cost are unchanged because the cryptographic setup and verification are identical to Groth16. □

Appendix C. Deferred Proofs for Security Analysis

Proof of 6.2 (Knowledge soundness). We show that any knowledge-soundness adversary against Sluice is also a knowledge-soundness adversary against standard Groth16, via the identity reduction.

1) **Setup.** The Setup algorithm is identical in both schemes (Algorithm 2): the CRS σ and the trapdoor τ_{trap} are generated by the same randomized algorithm. The only difference is the *storage layout* of the proving key on external media; this is invisible to the adversary, who receives only σ. Hence the CRS distribution is identical.
2) **Verification.** The Vfy algorithm is identical in both schemes. The verification equation
$$e([A]_1, [B]_2) = e([\alpha]_1, [\beta]_2) \cdot e([L_\phi]_1, [\gamma]_2) \cdot e([C]_1, [\delta]_2)$$
and the associated public-input check are unchanged. Therefore, an accepting proof for Sluice is also an accepting proof for standard Groth16.
3) **Reduction.** Let $\mathcal{A}$ be an adversary that produces an accepting proof π for Sluice with non-negligible probability. Define $\mathcal{A}' := \mathcal{A}$ (the identity transformation). Since $\mathcal{A}'$ receives the same CRS σ (identically distributed), and the verification check is the same, $\mathcal{A}'$ also produces an accepting proof for standard Groth16 with the same probability. By the knowledge-soundness of standard Groth16 [1, Theorem 2], there exists an extractor $\mathcal{X}$ for $\mathcal{A}'$; this extractor works equally well for $\mathcal{A}$.
4) **Conclusion.** The knowledge-soundness guarantee is inherited with no loss in the reduction.

□

Proof of Theorem 6.3 (Perfect zero-knowledge). The simulator Sim is the standard Groth16 simulator: given the trapdoor $\tau_{\mathrm{trap}} = (\alpha, \beta, \gamma, \delta, \tau)$, with $\gamma, \delta \in \mathbb{F}^*$, it samples random $A, B \leftarrow \mathbb{F}$ and computes C from the verification equation. Specifically, Sim sets
$$[A]_1 = A \cdot [1]_1, \qquad [B]_2 = B \cdot [1]_2,$$
$$[C]_1 = \tfrac{1}{\delta}\big(AB - \alpha\beta - \textstyle\sum_{i=0}^{\ell} a_i(\beta u_i(\tau) + \alpha v_i(\tau) + w_i(\tau))\big) \cdot [1]_1.$$

This simulator does not invoke the prover at all—neither the standard prover nor the streaming prover. Its output depends only on the trapdoor, the statement, and fresh randomness. Since Theorem 5.5 guarantees that the real proof distribution of Sluice equals that of standard Groth16 (given the same randomness), the indistinguishability argument of [1, Theorem 3] applies verbatim:
$$\{\mathsf{Sim}(R, \tau_{\mathrm{trap}}, \phi)\} \equiv \{\mathsf{Prove}^{\mathsf{std}}(R, \sigma, \phi, w)\} = \{\mathsf{Prove}^{\mathsf{rw}}(R, \sigma, \phi, w)\}.$$

Hence Sluice inherits perfect zero-knowledge. □

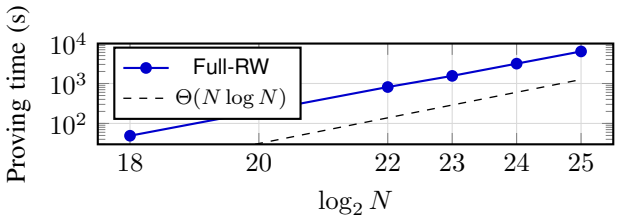
Appendix D. Artifact, Disclosures, and Supplementary Data

Generative AI use. Generative AI was used for editorial assistance and manuscript organization. All outputs were inspected by the authors for accuracy and originality; no generative AI output is used as an independent source of technical claims, proofs, implementation results, or experimental measurements.

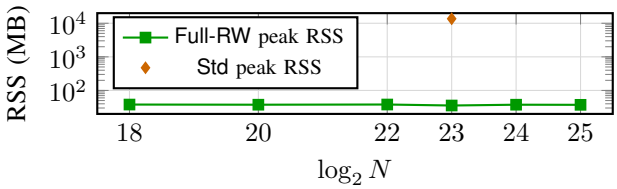
Data provenance. All measurements were generated by the Rust implementation. The artifact maps each aggregate to raw CSV summaries or capped-worker logs: direct scaling and throughput use `rw_rw_direct_scaling*.csv` and `storage_throughput_summary.csv`; capped rows use `cap_logs/*.csv`, `cap_logs/*.stderr`, `cap_logs/*.exit`, and `cap_logs/*.wall_s`; workload/component rows use the corresponding MiMC, arkworks, R1CS, NTT, and MSM CSVs; AEAD rows use `filevec_crypto_microbench_*.csv` and `rw_prove_only_aead_summary.csv`. The auxiliary mixed-commit plaintext repeat uses `rw_prove_only_23_repeat_summary.csv`. arkworks RSS is a setup+prove process high-water mark. The fixed-randomness proof-equivalence regression is reproduced by `cargo test --test streaming_e2e test_streaming_prove_n16 -- --exact`; it checks that the standard and streaming proof serializations are the same 128-byte Groth16 proof under fixed prover randomness.

Supplementary measurement notes. The fully streaming Full-RW prover produces verifier-accepted standard Groth16 proofs through $N = 2^{25}$. Three-run means at $N = 2^{18}, 2^{20}, 2^{22}$ are 49.2 s, 199.9 s, and 815.5 s; the single-run $N = 2^{23}, 2^{24}, 2^{25}$ points are 1 550.9 s, 3 139.8 s, and 6 343.5 s, with clean prove-only RSS in the 35.6–38.1 MB range. The $N = 2^{23}$ and $N = 2^{25}$ rows move 432.6 GiB and 1 872.3 GiB of application payload, giving effective `FileVec` throughput of 285.6 and 302.2 MiB/s; metadata and page-cache traffic are excluded. Storage-layer ChaCha20-Poly1305 `FileVec` microbenchmarks reach 216.7 MiB/s on 1 GiB (five-run mean) and 214.4 MiB/s on 25 GiB (single run), about 46% of plaintext throughput. Optional AEAD prove-only validation runs at $N = 2^{18}$ and $N = 2^{20}$ output valid 128-byte proofs; the $N = 2^{20}$ run takes 326.5 s with 39.5 MB peak RSS. They use a fixed benchmark key and do not claim production key management. A later-commit plaintext $N = 2^{23}$ repeat gives 1 644.3 s and 38.9 MB peak RSS; the mixed-commit two-run mean is 1 597.6 s with 66.0 s sample standard deviation, used only as auxiliary repeat evidence.

Capped and structured runs. The capped rows use Linux cgroup MaxRSS and are therefore reported separately from macOS clean-worker RSS. The standard capped worker reconstructs in-memory proving-key/QAP/witness state inside the cap, whereas Full-RW reopens sequential `FileVec` streams. On a BN-254 field-native MiMC-style Merkle structured workload, Std, Full-RW, and arkworks all output verifier-accepted 128-byte proofs. The structured-workload RSS entries are setup+prove high-water marks, not the clean prove-only memory metric used for Table 3. The plaintext prototype assumes trusted storage: normal drops unlink temporary streams, but crash leftovers, swap, journals, dumps, and VM/cloud snapshots are outside the theorem unless controlled by the prover operator or protected by authenticated stream encryption.



(a) Direct Full-RW prove-only time.

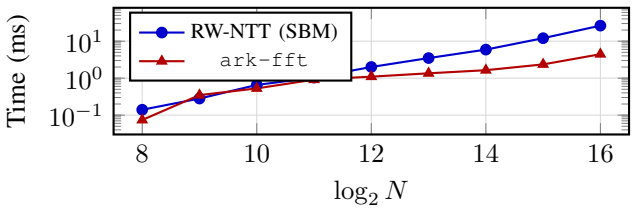


(b) Clean prove-only worker peak RSS.

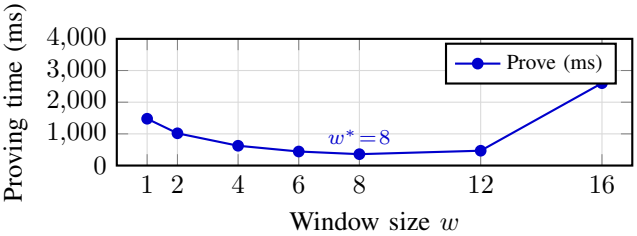
Figure 1. Scalability measurements through the demonstrated final-prover scale (BN-254, single-threaded). The $N = 2^{18}, 2^{20}, 2^{22}$ Full-RW points are three-run means; $N = 2^{23}, 2^{24}, 2^{25}$ are single runs due to duration. All Full-RW points produce valid 128-byte proofs. The Std memory point is the clean standard prove-only worker at $N = 2^{23}$.

$\log_2 N$	Runs	Full-RW prove	Peak RSS	R/W GiB
18	3	49.2 s	38.0 MB	5.6/5.2
20	3	199.9 s	37.6 MB	24.8/23.0
22	3	815.5 s	38.1 MB	108.1/101.0
23	1	1 550.9 s	35.6 MB	223.4/209.2
24	1	3 139.8 s	37.6 MB	468.0/439.6
25	1	6 343.5 s	37.3 MB	964.5/907.7

TABLE 4. FULL-RW DIRECT PROVE-ONLY SCALING. ROWS WITH THREE RUNS REPORT MEANS; LARGER ROWS ARE SINGLE RUNS DUE TO DURATION. ALL ROWS PRODUCE VALID 128-BYTE GROTH16 PROOFS.



(a) NTT: RW-NTT (SBM) vs. in-memory `ark-fft`.



(b) MSM window tradeoff ($N = 4096$).

Figure 2. Component-level analysis. (a) SBM-based RW-NTT incurs a multi-pass overhead relative to in-memory `ark-fft`, consistent with the $\mathcal{O}(\log N)$-pass I/O cost. (b) MSM proving time as a function of the Pippenger window size w; the minimum is at $w^* = 8$.

Table 5 gives the application-shaped structured workload and arkworks baseline. Table 6 separates the sparse R1CS streaming evaluation into multiply, sort, and reduce phases.

Variant	$\log_2 N$	Runs	Prove time	Setup+prove peak RSS
Std	20	3	31.1 ± 0.3 s	1 022.8 MB
Full-RW	20	3	116.5 ± 0.1 s	1 022.8 MB
arkworks	21	3	16.5 ± 0.1 s	$7\,340.2 \pm 230.5$ MB
arkworks	22	3	33.1 ± 0.7 s	$14\,013.1 \pm 85.5$ MB
arkworks	23	1	78.9 s	16 263.6 MB

TABLE 5. STRUCTURED MIMC-STYLE MERKLE WORKLOAD. THE RSS COLUMN IS SETUP+PROVE PROCESS HIGH-WATER RSS, NOT CLEAN PROVE-ONLY MEMORY. PROVE TIMES ARE MEANS WHEN MULTIPLE RUNS ARE AVAILABLE; ALL VARIANTS OUTPUT VERIFIER-ACCEPTED 128-BYTE PROOFS.

Type	N	Mult	Sort	Reduce	Total
Structured	1 024	0.02	0.06	0.06	0.13
Structured	4 096	0.06	0.25	0.26	0.57
Structured	16 384	0.24	1.00	0.97	2.21
Random	1 024	0.06	0.28	0.31	0.64
Random	4 096	0.24	1.29	1.46	2.99
Random	16 384	1.00	6.21	6.65	13.86

TABLE 6. R1CS STREAMING EVALUATION: PHASE BREAKDOWN (MS).

Interpretation. The R1CS evaluator spends most of its time in the sort/reduce phases, especially on random sparse matrices. This supports the design choice to treat sparse R1CS evaluation as an external-memory primitive rather than materializing dense vectors in RAM.