ZK-SPoW: ZK-Symbiotic Proof of Work

Anonymous*

February 2026 — Version 0.3 (Draft)

Abstract

Proof-of-work blockchains expend energy solely for network security. Proof of Useful Work (PoUW) attempts to reclaim this cost—a direction Ball et al. [1] show faces fundamental deployment constraints. ZK-SPoW (ZK-Symbiotic Proof of Work) inverts this relationship: useful ZK computation (STARK Merkle hashing) naturally produces PoW tickets as a mathematical byproduct. The construction operates Poseidon2 in compression function mode with extended width: in Symbiotic mode, every STARK Merkle hash accepts a child pair plus block header digest as input and simultaneously outputs a parent node advancing the ZK proof and multiple PoW tickets—from a single permutation. The miner does not choose the PoW input (nonce); the STARK computation determines it. Per-permutation usefulness is $U = t_0/t$ (t_0 useful elements in a width-t state); the remainder serves PoW integration. PoW mining (compute-bound) and STARK proving (memory-bound) share the same Poseidon2 hardware with zero switching overhead. The protocol is hardware-agnostic and applicable to any PoW chain adopting Poseidon2-based STARKs. We instantiate ZK-SPoW for Kaspa, replacing kHeavyHash with Width-24 Poseidon2 over the Mersenne field $\mathbb{F}_p$, $p = 2^{31} - 1$ (M31), achieving $U = 16/24 \approx 67\%$ with three tickets per permutation. Time-averaged usefulness $U_{\mathrm{avg}} = f_{\mathrm{sym}} \times U$ ranges from ~7% to ~40%, scaling with memory bandwidth (§A.5). A chain-specific STARK verifier supporting the extended width is required—a parameter change within Poseidon2's framework, not a new cryptographic construction. Security claims assume final Poseidon2 production parameters (§8).

1 Introduction

We develop ZK-SPoW as a general framework and instantiate it for **Kaspa**, a PoW blockchain achieving real-time decentralization (RTD) at 100 blocks per second via the DAG KNIGHT protocol [5]. Kaspa is evaluating StarkWare's Stwo—a high-performance STARK prover native to Poseidon2 over M31—making it a natural first candidate for PoW/STARK symbiosis.

1.1 The PoW Energy Problem

Kaspa uses kHeavyHash—cSHAKE256 (Keccak/SHA-3 family) composed with a 64×64 matrix multiplication over 4-bit nibbles—for proof of work. Like all traditional PoW schemes, the computational work produces no output beyond network security. The entire energy expenditure is justified solely by the security guarantees it provides.

1.2 The PoUW Paradox and Its Inversion

Ball et al. [1] formalize **Proof of Useful Work (PoUW)** as a PoW scheme where the mining computation simultaneously produces useful output. Their strict definition requires:

1. The PoW computation itself produces useful output

---

*Preprint. Work in progress.

2. The verifier can confirm the usefulness

3. The useful output is bound to the PoW evidence

The fundamental tension: PoW requires random exploration (nonce grinding), while useful computation requires specific, deterministic work. Prior PoUW constructions [1, 7, 8, 9] achieve provable security for specific problem classes, but require pre-hashing, SNARGs, or domain-specific verification that limits practical deployment in high-throughput blockchains (100 BPS).

ZK-SPoW inverts this relationship. Instead of making PoW results useful, we start from useful computation (STARK proof generation) and observe that PoW tickets emerge as a natural mathematical byproduct:

> **Conventional PoUW:** PoW computation → try to make results useful → fundamental constraints [1]

> **ZK-SPoW:** Useful ZK computation → PoW tickets as mathematical byproduct → no contradiction

Definition 1 (ZK-SPoW). *A PoW scheme where the hash function is a width-extended Poseidon2 compression function operating on STARK Merkle data, such that every permutation simultaneously advances a ZK proof and produces PoW tickets.*

The mechanism: STARK proof generation requires millions of Merkle hashes. Each Width-24 Poseidon2 Merkle hash takes $(left\_child, right\_child, header\_digest)$ as input and produces $(pow\_ticket_0, pow\_ticket_1, pow\_ticket_2)$ as output, where $pow\_ticket_0 = merkle\_parent$ simultaneously advances the ZK proof. All three output regions are checked against the difficulty target. The miner cannot choose the Merkle inputs—they are determined by the STARK computation. The "random exploration" required for PoW occurs naturally because STARK Merkle tree hashes are pseudorandom from the miner's perspective.

Definition 2 (Usefulness).
- $U = t_0/t = 16/24 \approx 67\%$ *when the ASIC is executing ZK proof computation (Symbiotic mode)*

- $U = 0\%$ *when the ASIC is grinding nonces without concurrent ZK computation (Pure PoW mode)*

The 33% overhead per permutation is the cost of PoW integration: 8 of 24 input state elements carry header_digest rather than ZK data. This is the inherent price of unifying ZK and PoW into a single permutation—not waste, but the cost of symbiosis. On the output side, all three 8-element regions serve as PoW tickets, and $S[0..7]$ simultaneously advances the ZK proof—the purest form of symbiosis.

U is a per-permutation metric. The time-averaged usefulness across all Poseidon2 cycles is $U_{avg} = f_{sym} \times U$, where f_{sym} is the fraction of cycles executing STARK Merkle hashes (determined by memory bandwidth; see §4.6 and §A.5).

If a PoW solution is found mid-proof, the block is submitted without interrupting ZK computation. When no ZK demand exists, the ASIC reverts to pure PoW ($U = 0\%$), identical to any conventional miner.

1.3   Stwo and Poseidon2

Kaspa is evaluating StarkWare's **Stwo** [4] as a potential STARK backend for verifiable programs (vProgs). Stwo operates over the Mersenne field M31 and uses **Poseidon2** [3] as its internal hash function for Merkle tree commitments and Fiat-Shamir challenges.

This creates a unique opportunity: if the PoW hash function is also Poseidon2 over M31, then the mining ASIC's primary computational element—the Poseidon2 pipeline—can serve both STARK proof generation and PoW mining. The cost is a Poseidon2 width extension from 16 to 24 elements with increased round count (+44–105% core area, $\sim + 22$–50% die area depending on implementation; see §3.2).

Stwo-Kaspa verifier. Standard Stwo uses Width-16 Poseidon2 in sponge mode. Width-24 Poseidon2 is a different cryptographic function (different MDS matrix, different round count, different S-box count per external round). ZK-SPoW therefore requires a Kaspa-specific verifier supporting Width-24 compression—a parameter change within Poseidon2's design framework [3], not a new cryptographic construction. Since the kHeavyHash $\rightarrow$ Poseidon2 transition already requires a hard fork with full-node verifier updates, the Width-24 adaptation is an incremental cost. This verifier is a prerequisite for Symbiotic mode; without it, the ASIC operates in Pure PoW mode only (§4.2).

1.4 Contributions

1. **PoUW paradox inversion.** We formalize ZK-Symbiotic Proof of Work, a construction where useful ZK computation (STARK Merkle hashing) naturally produces PoW tickets as a cryptographically bound byproduct—inverting the traditional PoUW direction explored by Ball et al. [1], Ofelimos [7], and Komargodski et al. [8, 9] (§1.2).

2. **Width-24 Poseidon2 parameterization.** We specify Width-24 Poseidon2 over M31 in compression function mode (1 permutation per Merkle hash, vs 2 in sponge mode) and verify its security parameters: $R_p = 22$ internal rounds for 128-bit security with $D = 5$, computed via Plonky3's round number formula [10] (§6.3).

3. **Complementary bottleneck architecture.** We demonstrate that PoW mining (compute-bound) and STARK proof generation (memory-bound) can share Poseidon2 hardware with zero-cycle switching overhead, and provide gate-level ASIC architecture analysis for a 7 nm implementation (§4, Appendix A).

Generality. The ZK-SPoW construction—width-extended Poseidon2 compression yielding PoW tickets as a STARK byproduct—is hardware-agnostic and not specific to Kaspa. Any PoW blockchain adopting Poseidon2-based STARKs can apply the same approach. We present Kaspa as the concrete instantiation throughout: its planned Stwo integration, existing ASIC mining ecosystem, and 100 BPS throughput make it a compelling first target. The ASIC analysis (Appendix A) demonstrates optimal-case efficiency on purpose-built hardware; the protocol functions identically on GPUs and FPGAs at proportionally lower throughput.

2 Notation and Definitions

Stwo baseline parameters (confirmed from source code [4]):

Note: All security claims in this paper assume final production round constants. The current Stwo implementation uses placeholder values (`EXTERNAL_ROUND_CONSTS` and `INTERNAL_ROUND_CONSTS` uniformly set to `1234`; see §8).

3 Protocol Specification

3.1 PoW Function Replacement

Current (kHeavyHash):

Symbol	Definition
$\mathbb{F}_p$	Finite field, $p = 2^{31} - 1$ (Mersenne prime M31)
Poseidon2$_\pi$	Poseidon2 permutation over $\mathbb{F}_p^t$
t	State width (number of field elements in permutation)
r	Rate: number of input/output elements (sponge mode)
c	Capacity: security parameter (sponge mode; hidden elements)
n	Hash output size in field elements ($n = 8$, giving 248 bits)
H	Block header (all consensus fields; see §3.4)
h_H	Header digest: PoseidonSponge(H excluding nonce) $\in \mathbb{F}_p^k$
k	Header digest element count ($k = 8$ for symmetric I/O and three PoW tickets)
(v_1, v_2)	Nonce: $v_1, v_2 \in \mathbb{F}_p^8$
T	Target $\in \mathbb{F}_p^8$ (difficulty-adjusted)
S	Poseidon2 state after permutation, $S \in \mathbb{F}_p^t$
U	Per-permutation usefulness: t_0/t (t_0 useful elements in width-t state)
f_{sym}	Fraction of Poseidon2 cycles executing STARK Merkle hashes
U_{avg}	Time-averaged usefulness: $f_{\text{sym}} \times U$

Table 1: Notation summary.

Parameter	Value
Field	M31, $p = 2^{31} - 1$
Hash output	8 elements = 248 bits
Standard width	$t_0 = 16$ (sponge mode: rate 8, capacity 8)
External rounds R_f	8 (4 + 4)
Internal rounds R_p	14
S-box exponent	$\alpha = 5$
Merkle hash	2 permutations per node (sponge: absorb left[8], absorb right[8])
Commitment hash	Blake2s (base layer), Poseidon2 (recursive proofs)

Table 2: Stwo baseline Poseidon2 parameters.

```
pre_pow_hash = Blake2b(H excluding nonce and timestamp)
inner        = cSHAKE256_PoW(pre_pow_hash || timestamp || nonce)
pow_hash     = cSHAKE256_Heavy(M * inner XOR inner)
valid iff pow_hash < target
```

where M is a 64×64 full-rank matrix over 4-bit nibbles (generated from `pre_pow_hash` via XoShiRo256++), nonce is 8 bytes (`u64`). The inner hash splits into 64 nibbles for matrix-vector multiplication; `cSHAKE256_PoW` and `cSHAKE256_Heavy` use domain strings `"ProofOfWorkHash"` and `"HeavyHash"` respectively [6].

Proposed (Poseidon2-PoW):

```
h_H = PoseidonSponge(H excluding nonce)    // amortized pre-hash (8 M31 elements)
S   = Poseidon2_pi(v1 || v2 || h_H)        // single permutation, width 24
pow_hash0 = (S[0], S[1], ..., S[7])         // 8 M31 elements = 248 bits
pow_hash1 = (S[8], S[9], ..., S[15])        // 8 M31 elements = 248 bits
pow_hash2 = (S[16], S[17], ..., S[23])      // 8 M31 elements = 248 bits
valid iff pow_hash0 < target OR pow_hash1 < target OR pow_hash2 < target
```

where (v_1, v_2) are 8 M31 elements each (64 bytes total nonce). The permutation operates in **compression function mode**—all 24 input elements are visible (no hidden capacity). This differs from Stwo's standard sponge mode (width 16, rate 8, capacity 8) but is a recommended

Poseidon2 usage mode [3]. Each permutation produces **three PoW tickets**—one from each 8-element output region. The comparison *pow_hash < target* interprets both as 248-bit unsigned integers via big-endian concatenation of 8 M31 elements, each zero-padded to 31 bits. In Symbiotic mode, $pow\_hash_0 = merkle\_parent$: the same output advances the ZK proof and serves as a PoW ticket (reading the value does not modify it).

Verification cost: One Poseidon2 permutation (width 24) + three target comparisons + one header pre-hash (amortized).

Standard PoW structure. ZK-SPoW is conventional hash-based PoW: miners explore a nonce space by computing Poseidon2 permutations and comparing outputs against a difficulty target, identically to Nakamoto-style PoW. The ZK component (Symbiotic mode, §4.1) is an optional revenue source sharing the same hardware—it does not modify the PoW function, security model, or difficulty adjustment. When no ZK demand exists, the ASIC reverts to conventional PoW mining (§4.2) with identical security guarantees.

3.2 Poseidon2 Width Extension

The core design change is extending the Poseidon2 permutation width to accommodate header digest as an additional input, and switching from sponge mode to compression function mode.

3.2.1 Stwo Baseline

Stwo's Poseidon2 operates in **sponge mode** with width 16:

```
Width t0 = 16
Rate  r  = 8    <- absorbs 8 elements per permutation
Capacity c = 8  <- hidden elements (security)
```

For Merkle tree commitments, each node hash requires **two sponge absorptions**:

```
Absorb left_child[8]  -> Permute -> (state updated)
Absorb right_child[8] -> Permute -> Squeeze output[8]
Total: 2 permutations per Merkle hash
```

3.2.2 Design Alternatives

Three approaches to integrate header digest into the Merkle hash:

Design	Width	Mode	Perm/Merkle	Perm/PoW	Core Δ	Die Δ
A: Header re-hash	16	Sponge	2	4	0%	0%
B: Width 20	20	Compression	1	1	+22%	∼+11%
C: Width 24	**24**	**Compression**	**1**	**1**	**+44–105%**	**∼+22–50%**

Table 3: Design alternatives for header digest integration.

Design A: No Poseidon2 modification. Standard width-16 sponge Merkle hash (2 permutations: absorb *left_child*, absorb *right_child*). To bind the header for PoW, a *second* sponge hash is required: sponge(*merkle_parent*, *header_digest*) → *pow_hash* (2 additional permutations). Total: **4 permutations per PoW draw**, vs 1 for compression function mode. The STARK tree itself is unmodified (2 perm/node), but each PoW draw doubles the permutation cost.

Design B: Extend to width 20 with *header_digest*[4]. Compression function mode, 1 permutation per Merkle hash and per PoW draw. However, header digest is only 4 M31 elements (124 bits), giving a birthday collision bound of 2^{62}—well below the 128-bit security target. Asymmetric 8+8+4 I/O with 4 wasted output elements.

Design C (selected): Extend to width 24 with *header_digest*[8]. Symmetric 8+8+8 I/O—header digest is baked into every Merkle hash, so each permutation simultaneously advances the STARK tree and produces a PoW-checkable output. **1 permutation per Merkle hash and per PoW draw**, with zero additional overhead. $S[0..7]$ serves as both *merkle_parent* (STARK) and PoW output. Header digest: 248 bits (birthday bound 2^{124}). Width 24 is within the Poseidon2 paper's analyzed parameter range [3]. **ZK throughput is unaffected**: STARK Merkle hashing is SRAM-bandwidth-bound regardless of core width (see §4.6). The width extension cost manifests as $U = 16/24 \approx 67\%$ (§1.2). **Requires Stwo-Kaspa verifier** supporting Width-24 compression function mode.

Design A vs C tradeoff. Design A requires no Stwo modification and uses well-analyzed Width-16 parameters. However, each PoW draw costs 4 permutations (2 Merkle + 2 header binding), while Design C completes both Merkle hash and PoW draw in a single permutation. Accounting for Design A's smaller cores (more cores per die), Design C still yields $\sim 2\times$ higher aggregate PoW throughput per die. Design C also simplifies scheduling (each permutation is stateless, vs sponge state tracking). The cost: Design C requires a Width-24 Stwo-Kaspa verifier (§8) and +44–105% core area overhead.

3.2.3 Proposed Extension (Design C)

```
Standard Stwo:    Width t0 = 16,  Sponge (rate 8, capacity 8)
Proposed ZK-SPoW: Width t  = 24,  Compression function (all 24 visible)
```

Impact on Poseidon2 internals:

Component	Width 16 (standard)	Width 24 (proposed)	Change
External MDS	4 M4 blocks	6 M4 blocks	+50% additions
Internal MDS	16 multiplications	24 multiplications	+50% mult.
S-box (external)	16 per round	24 per round	+50%
S-box (internal)	1 per round	1 per round	unchanged
Internal rounds R_p	14	22	+57%
Total rounds	22 (8+14)	30 (8+22)	+36%
S-box operations	142	214	+51%
State registers	$16 \times 31 = 496$ bits	$24 \times 31 = 744$ bits	+50%

Table 4: Poseidon2 internals: Width-16 vs Width-24.

Internal round MDS scales as $O(t)$, not $O(t^2)$, because the sparse MDS structure (diagonal + rank-1) requires only t multiplications per round. This makes the width extension significantly cheaper than for standard Poseidon.

Core area overhead: $+44\%$ **(datapath width) to** $\sim +105\%$ **(fully pipelined).** The datapath widens by 50% (24/16), and Width-24 requires $R_p = 22$ internal rounds vs Width-16's $R_p = 14$ (§6.3), adding +36% pipeline depth (30 vs 22 total rounds). In an iterative (round-reuse) design, core area increases by $\sim +50\%$ (width only) with 36% more cycles per hash. In a fully pipelined design, core area approximately doubles. Total die area impact: $\sim +22\%$ to $\sim +50\%$ depending on implementation (Poseidon2 is 50% of die; see Appendix A). The usefulness cost is $U = t_0/t = 16/24 \approx 67\%$—the 33% overhead per permutation serves PoW integration, not ZK computation.

3.2.4 Compression Function vs Sponge

Standard Stwo uses sponge mode with 8 hidden capacity elements. This proposal uses **compression function mode** where all 24 state elements are visible.

Property	Sponge (Stwo standard)	Compression (ZK-SPoW)
Hidden state	8 capacity elements	None (all visible)
Security model	Indifferentiability	Collision/preimage of π
Perm. per Merkle hash	2	**1**
Width	16	24
PoW tickets per hash	0	**3**

Table 5: Sponge vs compression function mode.

Both modes are established Poseidon2 usage modes [3]. The Poseidon2 paper recommends compression function mode for Merkle trees, noting up to $5\times$ efficiency over sponge mode in compute-bound settings. On ASIC implementations, STARK Merkle throughput is typically SRAM-bandwidth-bound (see §A.5), so the per-permutation efficiency gain manifests as more Poseidon2 cycles available for PoW rather than faster ZK proof generation. Width 24 is within the paper's analyzed parameter range [3, 10].

3.3 I/O Mapping

For a single Poseidon2 permutation with state $S \in \mathbb{F}_p^{24}$ (compression function mode):

```
INPUT (24 = 8+8+8, all visible):
  S[0..7]   <- left_child     8 M31 elements (248 bits)
  S[8..15]  <- right_child    8 M31 elements (248 bits)
  S[16..23] <- header_digest  8 M31 elements (248 bits)

        [ Poseidon2 permutation (t = 24) ]
        [ R_f = 8 external + R_p = 22 int ]

OUTPUT (24 = 8+8+8, all visible):
  S[0..7]   -> pow_ticket0    merkle_parent (STARK) AND PoW ticket 0 (248 bits)
  S[8..15]  -> pow_ticket1    PoW ticket 1 (248 bits)
  S[16..23] -> pow_ticket2    PoW ticket 2 (248 bits)
```

Symmetric 8+8+8 I/O with no unused output elements. No capacity elements—compression function mode exposes all state elements. Security relies on the Poseidon2 permutation's collision resistance and PRP properties (§6.3). $S[0..7]$ serves dual roles: it advances the STARK Merkle tree (as *merkle_parent*) and is checked against the PoW target. Reading $S[0..7]$ for PoW comparison does not modify the value used by the STARK computation.

Note on full diffusion: The Poseidon2 permutation mixes all 24 state elements through its MDS matrix every round. All output elements are functions of all input elements—there is no structural separation between output regions. The 8+8+8 labeling is a convention for reading the output, not a property of the permutation. This means *merkle_parent* depends on *header_digest*, and the Stwo-Kaspa verifier must account for this.

Dual-use property:

Mode	$S[0..7]_{\mathrm{in}}$	$S[8..15]_{\mathrm{in}}$	$S[16..23]_{\mathrm{in}}$	$S[0..7]_{\mathrm{out}}$	$S[8..15]_{\mathrm{out}}$	$S[16..23]_{\mathrm{out}}$
Symbiotic	left child	right child	h_H	merkle parent + ticket$_0$	PoW ticket$_1$	PoW ticket$_2$
Pure PoW	v_1	v_2	h_H	PoW ticket$_0$	PoW ticket$_1$	PoW ticket$_2$

Table 6: Dual-use I/O mapping.

The same Poseidon2 hardware computes both modes. Only the input source for $S[0..15]$ differs

(SRAM Merkle data vs random nonces). $S[16..23]$ is always h_H. **Each permutation produces 3 PoW tickets**—one from each 8-element output region.

Note on ticket granularity. The 3×8-element partition is a protocol convention, not a cryptographic constraint. Under PRP, any contiguous or non-overlapping subset of the 24 output elements is pseudorandom. The default 8-element reading (248 bits) provides sufficient difficulty-target precision at current network scale and matches Stwo's hash output convention. If future hashrate growth demands finer target granularity, the comparison window can be widened—up to the full 24-element output (744 bits)—without modifying the hash function; only the target comparison logic changes.

3.4   Block Structure

Proposed header (change from current Kaspa in bold):

```
Header {
  version:                u16
  parents_by_level:       [[Hash]]       // DAGKnight multi-level parents
  hash_merkle_root:       Hash           // transaction Merkle root
  accepted_id_merkle_root: Hash
  utxo_commitment:        Hash
  timestamp:              u64            // milliseconds
  bits:                   u32            // difficulty target
  nonce:                  [F_p; 16]      // **64 bytes (currently u64 = 8 bytes)**
  daa_score:              u64
  blue_work:              BlueWorkType
  blue_score:             u64
  pruning_point:          Hash
}
```

The only structural change is the nonce expansion from `u64` (8 bytes) to `[F_p; 16]` (64 bytes, +56 bytes per block, $\sim 0.04\%$ of 125 KB). The nonce maps to Poseidon2 input positions $S[0..15]$ as (v_1, v_2), each 8 M31 elements.

	kHeavyHash (current)	Poseidon2-PoW (proposed)
Nonce	`u64` (8 bytes)	`[F_p; 16]` (64 bytes)
PoW function	kHeavyHash $\rightarrow$ 256-bit	Poseidon2 Width-24 $\rightarrow$ 248-bit
Block hash	Blake2b-256 $\rightarrow$ 256-bit	Blake2b-256 $\rightarrow$ 256-bit (**unchanged**)

Table 7: Block structure comparison.

Block hash vs PoW hash. Kaspa computes block identity and PoW with separate hash functions. The block hash (Blake2b-256 over the full serialized header including nonce) provides DAG references and block identification—unchanged by this proposal. Only the PoW function is replaced: $h_H = \text{PoseidonSponge}(\text{header excluding nonce})$ is the amortized pre-hash, combined with the nonce in Width-24 Poseidon2 (§3.1). The 248-bit PoW output does not affect the 256-bit block hash.

STARK proofs are NOT included in the block. They are submitted as independent transactions in the mempool, providing economic value to the ZK ecosystem (vProgs fees). This eliminates the +3–5 MB/s bandwidth overhead that would result from mandatory per-block STARK proofs at 100 BPS.

3.5 Header Digest Collision Resistance

The header digest h_H compresses the block header (excluding nonce) into k field elements. If k is too small, an attacker can find two headers $H_A \neq H_B$ with identical h_H, allowing PoW solutions to be transplanted between conflicting blocks.

k (elements)	Bits	Birthday bound	Security
1	31	$2^{15} \approx 32\text{K}$	INSECURE
2	62	$2^{31} \approx 2 \times 10^9$	INSECURE
4	124	$2^{62} \approx 4.6 \times 10^{18}$	SECURE
8	248	2^{124}	Conservative but justified

Table 8: Header digest collision resistance by element count.

Minimum: $k = 4$ (124-bit collision resistance). We choose **k = 8** (248-bit, matching PoW hash size) to enable symmetric 8+8+8 I/O and **3** PoW tickets per permutation. The triple-ticket structure and high permutation rate ($\sim 10^9$/sec per ASIC) make the conservative choice appropriate. Width extension: $t = 16 + 8 = 24$.

4 Operating Modes

4.1 Symbiotic Mode (Stwo Prover Active)

When ZK proof demand exists, the ASIC runs the Stwo prover. The STARK proof generation pipeline:

```
1. Trace generation      -> circuit evaluation
2. NTT (LDE)             -> polynomial domain extension
3. Merkle tree           -> Poseidon2 hashing (PoW tickets appear here)
4. Fiat-Shamir challenge -> derived from Merkle root
5. FRI rounds            -> folding + commitment
```

At step 3, every Merkle hash is:

```
S = Poseidon2_pi(n_L || n_R || h_H)   // width 24, compression function
merkle_parent = S[0..7]     <- advances STARK proof AND checked as PoW ticket 0
pow_ticket1   = S[8..15]    <- checked against PoW target
pow_ticket2   = S[16..23]   <- checked against PoW target

if S[0..7] < target OR S[8..15] < target OR S[16..23] < target -> BLOCK FOUND
```

Every Poseidon2 invocation in the STARK computation simultaneously: **(a)** advances the ZK proof (economic value—the useful work that drives the permutation), and **(b)** produces PoW tickets (network security—mathematical byproduct of the same permutation).

The miner does not choose the Merkle inputs (n_L, n_R)—they are determined by the STARK computation. The "random exploration" required for PoW occurs naturally because STARK Merkle tree hashes are pseudorandom from the miner's perspective.

$U = t_0/t = 16/24 \approx 67\%$. The 33% usefulness gap is the width extension overhead: 8 of 24 input state elements carry *header_digest* rather than ZK data. There is no "ZK only" mode—every Width-24 permutation produces PoW tickets, regardless of input.

Header digest and Merkle tree. Because Poseidon2's MDS matrix provides full diffusion, the *merkle_parent* output depends on *header_digest*. This means the STARK Merkle tree is bound to a specific block header. At $\sim$2G hash/sec, a complete Merkle tree builds within one

block interval (10 ms at 100 BPS). The Stwo-Kaspa verifier reconstructs Merkle nodes using the same Width-24 compression with the known *header_digest*.

Header freshness. A STARK proof spans multiple Merkle commitment phases (step 3 and FRI rounds)—typically $O(10)$ phases—and takes seconds to complete (dominated by NTT and trace generation). The *header_digest* is fixed per Merkle tree phase; each phase completes within one block interval (< 10 ms at $\sim$2G hash/sec with $\sim$2.08G hash throughput per phase). Between phases, the *header_digest* register updates to the current block. PoW tickets from each phase are valid for that phase's header. Maximum staleness is 1 block (10 ms): a PoW ticket found at the end of a Merkle phase references a *header_digest* that is at most one block behind the DAG tip. In DAGKnight's DAG structure [5], this is well within tolerance—DAGKnight accepts blocks with parent sets up to k blocks deep (where k is the anticone parameter, typically $k \geq 10$ at 100 BPS), and 1-block staleness is indistinguishable from normal parallel block production.

4.2 Pure PoW Mode (No ZK Demand)

When no ZK proofs are requested:

```
loop:
  v1 = next_nonce_1()
  v2 = next_nonce_2()
  S = Poseidon2_pi(v1 || v2 || h_H)   // width 24, compression function
  if S[0..7] < target OR S[8..15] < target OR S[16..23] < target -> BLOCK FOUND
```

Identical Poseidon2 pipeline, identical throughput. The only difference is the input source: random/sequential nonces instead of STARK Merkle children.

$U = 0\%$. No ZK proof is being computed. The ASIC provides network security only, equivalent to any conventional PoW miner. This is not waste—security has value—but it is not useful work in the ZK-SPoW sense.

4.3 Linear Mode Transition

```
+-- Poseidon2 Pipeline (always 100% utilized) ----------+
|                                                       |
|  Input MUX (per-cycle decision):                      |
|    SRAM data ready  -> STARK Merkle hash (Symbiotic)  |
|    SRAM not ready   -> PoW nonce hash     (PoW)       |
|                                                       |
|  Switching cost: 0 cycles (combinational MUX, ~300 G) |
|  Hashrate: invariant across all modes                 |
+-------------------------------------------------------+
```

The transition between Symbiotic and Pure PoW is **per-cycle and linear**, not a discrete mode switch. When the Poseidon2 pipeline is executing STARK computation, that cycle is Symbiotic. When SRAM data is not ready, a random nonce hash is substituted and that cycle is PoW. The pipeline is always full—the ratio of Symbiotic to PoW cycles is determined by SRAM bandwidth, not by difficulty or protocol parameters.

4.4 Hashrate Invariance

Proposition 1. *Total PoW hashrate $\mathcal{H}$ is independent of the operating mode.*

Argument. $\mathcal{H} = N_{\text{cores}} \times$ throughput_per_core. Each core's throughput is 1 hash per pipeline_depth cycles (fully pipelined), regardless of whether the input is a STARK Merkle pair or a random

nonce. Input MUX adds zero latency (combinational logic). Therefore $\mathcal{H}$ is constant across Symbiotic, Pure PoW, and any mixed state. □

4.5 Difficulty Independence

U is determined by ZK demand and width ratio, not by PoW difficulty.

Condition	U	Rationale
Stwo Prover active, any difficulty	$\approx 67\%$	ZK proof, minus width overhead
No ZK demand, any difficulty	0%	Pure PoW = security only

Table 9: Usefulness is independent of difficulty.

Difficulty affects how many hashes are needed to find a block, but does not change U. Whether difficulty is 8M or 188G, the ASIC either computes ZK proofs ($U \approx 67\%$) or grinds nonces ($U = 0\%$). The ratio of STARK-to-PoW cycles within the pipeline is determined by SRAM bandwidth (a hardware constant), not by the network's difficulty target.

4.6 Complementary Bottleneck Structure

The simultaneous execution of PoW and STARK is possible because they bottleneck on different resources:

Resource	PoW	STARK	Combined
Poseidon2 cores	**100%** (compute-bound)	Low (SRAM-starved)	$\sim$**100%**
NTT unit	0%	**100%**	**100%**
SRAM bandwidth	0% (registers only)	**100%** (memory-bound)	**100%**

Table 10: Complementary bottleneck structure.

PoW is compute-bound (limited by Poseidon2 throughput). STARK is memory-bound (limited by SRAM bandwidth feeding Merkle data to Poseidon2). They share Poseidon2 cores but contend on different bottleneck resources, achieving near-perfect utilization of all hardware components simultaneously. Under 200 GB/s SRAM bandwidth, the STARK allocation is $f_{\text{sym}} \approx 10\%$ of Poseidon2 cycles, yielding $U_{\text{avg}} = f_{\text{sym}} \times U \approx 6.7\%$ time-averaged usefulness (see §A.5 for derivation). This complementary structure is the foundation of the economic analysis in §5.

Width-24 efficiency. Width-24 compression uses 1 permutation per Merkle hash versus Width-16 sponge's 2 permutations. This halves STARK's Poseidon2 cycle consumption, freeing more cycles for PoW. The STARK proof generation rate itself remains SRAM-bandwidth-bound (see §A.5 for quantitative analysis).

5 Pareto Analysis

5.1 Competing Designs

Four designs compared under identical die area and power budget:

Pure PoW achieves $\sim$1.9$\times$ hashrate on a die-area basis (95%/50% Poseidon2 allocation) but produces no ZK proofs ($U = 0\%$). On a hashes-per-watt basis, the gap narrows to $\sim$1.1–1.2$\times$ because idle NTT and SRAM contribute static leakage ($\sim$10–20% of dynamic power; varies by

Design	Die allocation	Hashrate	ZK	U	Mine?
Pure PoW	95% Pos2, 5% ctrl	$\sim 1.9\,\mathcal{H}$	0	0%	Yes
Pure Stwo	20% Pos2, 40% NTT, 35% SRAM	0	$Z_{\max}$	100%	No
ZK-SPoW	**50% Pos2, 25% NTT, 20% SRAM**	$\mathcal{H}$	Z	$\approx\mathbf{67}$%	**Yes**
ZK-SPoW+HBM	50% Pos2, 25% NTT, 20% HBM	$\mathcal{H}$	Z_{hbm}	≈ 67%	Yes

Table 11: Pareto comparison of competing designs.

process node). Pure Stwo uses standard Width-16 cores ($U = 100\%$) but cannot mine. ZK-SPoW achieves $U \approx 67\%$ and mining capability—the 33% usefulness gap is the cost of PoW integration via width extension.

5.2 Economic Dominance

Difficulty adjusts to total network hashrate. When all N miners use the same design, per-ASIC mining revenue is B/N regardless of absolute hashrate. The differentiator is ZK revenue:

$$\text{ZK-SPoW revenue} = B/N + Z \cdot F \; > \; B/N = \text{Pure PoW revenue} \quad \text{(for any } ZF > 0)$$

Pure PoW's ~ 1.1–$1.2\times$ power-efficiency advantage yields at most ~ 10–20% more mining revenue per watt in a mixed network. Once ZK fee income $Z \cdot F$ exceeds this margin, ZK-SPoW dominates. The crossover point depends on Kaspa's ZK market development, network growth trajectory, and adoption dynamics—detailed economic modeling is required for quantitative predictions.

PoW security without ZK demand. ZK-SPoW does not condition PoW security on ZK demand. When $Z \cdot F = 0$ (no ZK market), the ASIC operates as a conventional PoW miner with a hash/watt disadvantage due to the die area overhead (§3.2.3). Difficulty adjustment absorbs this: in a homogeneous ZK-SPoW network, per-ASIC mining revenue is identical to a homogeneous Pure PoW network. The die overhead is the cost of optionality—it purchases the ability to capture ZK revenue when the market emerges, without sacrificing PoW security in the interim.

6 Security Considerations

6.1 Poseidon2 Cryptographic Properties

Poseidon2 [3] provides 128-bit security in sponge mode with capacity $c = 8$ M31 elements (248 bits). In compression function mode (ZK-SPoW), security relies on collision resistance and preimage resistance of the permutation—well-established properties within the Poseidon2 framework [3]. Note that [3] explicitly recommends compression function mode for Merkle trees (Section 4.2 of [3]), with security analysis independent of the sponge indifferentiability framework. Known algebraic attack vectors (Gröbner basis, interpolation, differential) have been analyzed for standard parameters. StarkWare has adopted Poseidon2 for production use in Starknet, representing significant implicit endorsement of its security.

Note: All security claims in this paper assume final production round constants. The current Stwo implementation uses placeholder values (see §8).

6.2 Single Primitive Dependency

Risk: A cryptographic break in Poseidon2 compromises both PoW security and STARK validity simultaneously. This concentration risk is shared with the broader Poseidon2 ecosystem (notably Starknet).

Component	Current Kaspa	Proposed
PoW	kHeavyHash (Blake2b + cSHAKE256)	Poseidon2
STARK	Poseidon2	Poseidon2
Independence	PoW $\neq$ STARK	PoW = STARK

Table 12: Single primitive dependency.

6.3 Width-24 Security

Collision resistance (STARK binding). STARK Merkle tree integrity requires collision resistance of the Width-24 compression function. With 248-bit output (8 M31 elements), the birthday bound is 2^{124}, providing 124-bit collision security. Width-24 provides strictly more state than Width-16 (24 vs 16 elements) and is within [3]'s analyzed parameter range.

STARK soundness. The *header_digest* acts as a fixed salt in each Merkle hash: it is determined by the block header before proof generation begins and cannot be chosen adaptively by the prover. STARK soundness therefore reduces to collision resistance of the Width-24 compression function with a fixed third input—a strictly easier assumption than collision resistance under adversarially chosen inputs. The Stwo-Kaspa verifier reconstructs Merkle nodes using the same *header_digest*, preserving the binding property.

Preimage resistance (PoW security). PoW requires only preimage resistance of the 248-bit output—trivially satisfied by 30 rounds of Poseidon2 (8 external + 22 internal).

R_p **for Width-24.** The internal round count increases from $R_p = 14$ (Width-16) to $R_p = 22$ (Width-24) for 128-bit security at $D = 5$ over M31. This is computed via Plonky3's round number formula [10], which applies the security constraints from [2, 3] plus the algebraic attack bound from Khovratovich et al. (ePrint 2023/537), with a security margin of $R_f \mathrel{+}= 2$, $R_p \times 1.075$. The binding constraint is statistical ($R_f \geq 6$). Total rounds: 30 (8 + 22) vs 22 (8 + 14) for Width-16. S-box operations per permutation increase by 51% (214 vs 142); however, compression function mode requires only 1 permutation per Merkle hash (vs 2 in sponge mode), yielding 25% fewer S-boxes per hash (214 vs $2 \times 142 = 284$). Supplementary verification: M_I^k is invertible for all $k = 1..48$ (necessary condition for subspace trail resistance [3]). Diffusion analysis confirms full input-to-output dependency from the first external round. Algebraic degree after the full 30-round permutation exceeds 2^{69}, well above the 2^{64} threshold for interpolation security at 128 bits.

6.4 PoW Hash Distribution

All three output regions—$pow\_ticket_0 = S[0..7]$, $pow\_ticket_1 = S[8..15]$, and $pow\_ticket_2 = S[16..23]$—are outputs of the same Poseidon2 permutation. In compression function mode, all 24 state elements are visible by design. Security relies on the permutation's PRP properties: given a random-looking input, all output elements should be indistinguishable from random. The three PoW tickets are deterministically linked (same permutation), but each is individually pseudorandom. An attacker who could predict one ticket from another without computing the full permutation would violate the PRP assumption—equivalent to breaking Poseidon2. The full-round permutation (8 external + 22 internal) ensures all output elements are cryptographically mixed across all 24 state positions.

Triple-ticket mining: Under the PRP assumption on Poseidon2, the three tickets are mutually independent (Appendix B.7). With three 248-bit tickets per permutation, the per-permutation

success probability is exact:

$$P(\text{valid}) = 1 - (1 - p)^3 = 3p - 3p^2 + p^3, \quad p = T/2^{248}[1]$$

Note that the number of tickets per permutation does not affect mining economics in a homogeneous network: difficulty adjustment absorbs any change in per-permutation success probability, leaving per-miner revenue at B/N (§5.2). The triple-ticket structure is a natural consequence of Width-24's symmetric 8+8+8 output and Stwo's 8-element hash convention, not a hashrate optimization.

6.5 Quantum Resistance

Poseidon2's security against quantum adversaries:

- Grover's algorithm halves the effective hash bits: $248/2 = 124$-bit quantum security

- Comparable to SHA-256 under quantum attack ($256/2 = 128$ bits)

- kHeavyHash: $256/2 = 128$-bit quantum security

- **Delta:** -8 bits classical (248 vs 256) / -4 bits quantum (124 vs 128) from the transition. Both values remain well above the 100-bit security floor considered acceptable for PoW functions.

7 Comparison with Prior Work

7.1 Nockchain (zkPoW)

Nockchain [11] is a Layer-1 blockchain using Zero-Knowledge Proof of Work (zkPoW), launched in May 2025. Miners compute a ZK proof of a deterministic puzzle (populating a binary tree with Goldilocks field elements), then hash the proof; the hash must meet a difficulty target. This is the closest deployed system to ZK-SPoW.

	Nockchain (zkPoW)	ZK-SPoW
PoW hash	hash(ZK proof) $< T$	Poseidon2 output $< T$
PoW hash generation	Two-step: ZK proof $\rightarrow$ external hash	**Single-step**: STARK Merkle hash = PoW hash
ZK overhead	100% (every cycle is ZK)	$\sim$10% (SRAM-bandwidth-limited)
Field / output	Goldilocks (64-bit), 256-bit hash	M31 (31-bit), 248-bit output
Hardware target	GPU	ASIC-optimized
Useful work	ZK proof of puzzle	STARK Merkle hashing
STARK enforcement	Mandatory (proof = block)	Market-driven

Table 13: Nockchain vs ZK-SPoW.

The key architectural difference: in Nockchain, the ZK proof is computed first, then hashed externally for PoW—two disjoint steps. In ZK-SPoW, the internal STARK Merkle hash *directly* produces the PoW output from a single Poseidon2 permutation, with zero additional overhead. This tight coupling enables the complementary bottleneck structure (§4.6): PoW fills idle Poseidon2 cycles while STARK is memory-bound, whereas Nockchain's sequential proof-then-hash architecture does not exploit resource complementarity.

Nockchain mandates ZK proofs in every block. ZK-SPoW takes a market-driven approach (no mandatory proofs; ZK adoption through economic incentives), preserving Kaspa's bandwidth profile at 100 BPS.

---

[1]Each M31 element ranges over $[0, 2^{31} - 2]$, so the exact denominator is $(2^{31} - 1)^8$ rather than 2^{248}. The ratio $(2^{31} - 1)^8/2^{248} = (1 - 2^{-31})^8 \approx 1 - 3.7 \times 10^{-9}$. We use 2^{248} throughout as a convenient approximation.

7.2 Relationship to Ball et al.

Ball et al. [1] formalize PoUW in the direction **PoW → useful output**. ZK-SPoW operates in the inverse direction: **useful computation → PoW output**.

Ball et al. criterion	PoW → useful (their framework)	Useful → PoW (ZK-SPoW)
Produces useful output	Partial: PoW-fill = security-only	**Yes**: every Symbiotic perm
Verifier confirms	Not enforced (Option C)	STARK proofs publicly checkable
Bound to PoW	Not enforced	**Inherent**: same permutation

Table 14: Relationship to Ball et al. [1].

Under their original framework (PoW → useful), ZK-SPoW satisfies 0 of 3 criteria strictly. Under the inverted framing (useful → PoW), the evaluation changes: the STARK computation drives the permutation, PoW tickets are a cryptographically bound byproduct, and the proof is publicly verifiable.

The deeper point: Ball et al.'s hardness results [1] constrain the PoW → useful direction. ZK-SPoW sidesteps these constraints by never attempting to make PoW useful. Instead, useful computation (STARK proving) happens to produce PoW-valid outputs because Poseidon2's pseudorandom outputs naturally fall below the target at the expected rate.

Ofelimos [7] is the closest prior work, using SNARK proofs as useful work within a provably secure PoUW framework. Komargodski et al. [8, 9] explore PoUW via matrix multiplication and external utility functions. ZK-SPoW differs from both: (1) the useful computation is market-driven rather than protocol-mandated (Option C), and (2) PoW tickets emerge as a byproduct of STARK hashing rather than through a separate verification mechanism.
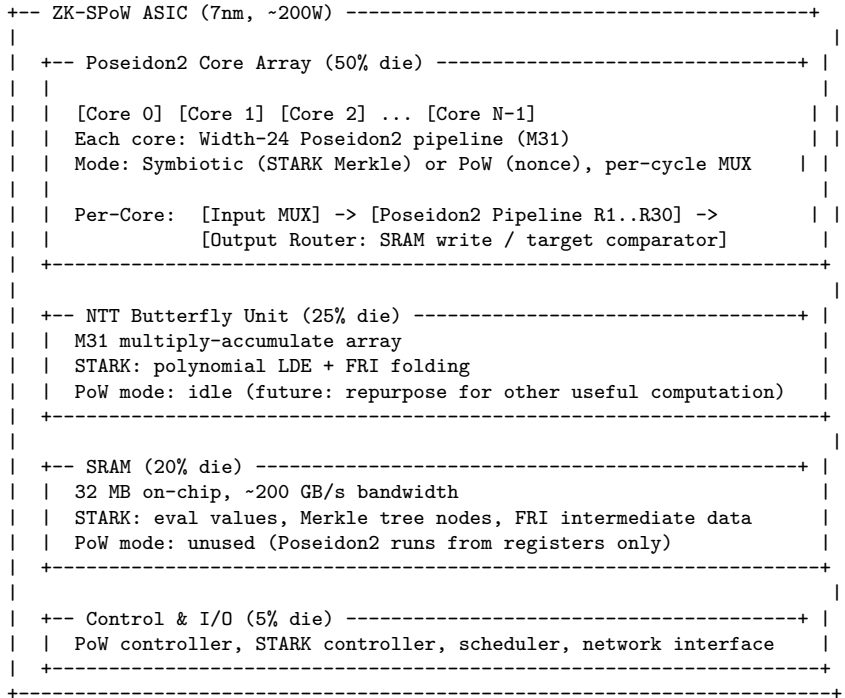
8 Open Questions

1. **Stwo production parameters.** The baseline parameters (Width 16, Rate 8, Capacity 8, $R_f = 8$, $R_p = 14$) are confirmed from source code [4]. However, round constants are explicit placeholders (`1234`; see TODO in source). Final production constants are required before security analysis can be completed.

2. R_p **for Width-24 (resolved; pending independent verification).** $R_p = 22$ computed via Plonky3's round number formula [10]; independent cryptanalytic verification pending. See §6.3.

3. **Stwo-Kaspa verifier.** Width-24 Poseidon2 compression is a different cryptographic function from Width-16 sponge (different MDS matrix, different round count). A Kaspa-specific verifier is required. Plonky3 [10] already provides a production Width-24 Poseidon2 implementation over M31 with verified parameters: external MDS = $\mathrm{circ}(2M_4, M_4, \ldots, M_4)$; internal MDS = $\mathbf{1}\mathbf{1}^\top + \mathrm{diag}(V)$ with $V = [-2, 1, 2, 4, \ldots, 2^{22}]$; $R_f = 8$, $R_p = 22$. The remaining requirements are: (a) round constant finalization (Grain LFSR or PRNG; both Stwo and Plonky3 currently use non-production constants), (b) integration with Stwo's proof system, (c) independent security certification. StarkWare's willingness to accept Width-24 as an upstream configuration option determines whether this is a lightweight fork or a permanent maintenance burden.

4. **Hard fork governance.** Transitioning from kHeavyHash to Poseidon2 renders existing kHeavyHash ASICs obsolete and requires community consensus.

5. **Mining pool protocol.** The 64-byte nonce (vs current 8-byte) requires updates to the stratum protocol for nonce range distribution. Each nonce element must satisfy $0 \leq v_i <$

$2^{31} - 1$ (valid M31 range), adding a validity constraint absent in conventional 64-bit nonce mining. Stratum V2 may accommodate this natively.

6. **ZK market maturity.** The economic advantage of ZK-SPoW over Pure PoW depends on sufficient ZK proof demand. The timeline for this market to develop is uncertain and requires dedicated economic modeling.

7. **Complementary bottleneck validation.** The claim that PoW (compute-bound) and STARK (memory-bound) can run simultaneously at full throughput requires hardware-level validation on actual ASIC designs.

8. **Triple-ticket independence (resolved under PRP).** Under the PRP assumption, for any fixed input x, the permutation output $\pi(x)$ is uniformly distributed over $\mathbb{F}_p^{24}$. A uniform distribution on the product space $\mathbb{F}_p^8 \times \mathbb{F}_p^8 \times \mathbb{F}_p^8$ implies that $S[0..7]$, $S[8..15]$, $S[16..23]$ are mutually independent. The joint success probability $q = 1 - (1 - p)^3 = 3p - 3p^2 + p^3$ is therefore exact. Any detectable correlation would constitute a PRP distinguisher—equivalent to breaking Poseidon2. See Appendix B.7.

9. **Trace grinding (resolved under PRP).** Under the PRP assumption on Poseidon2, trace selection does not affect the PoW ticket success distribution. The total number of permutations across all STARK commitment phases (initial Merkle tree plus FRI rounds) is determined by protocol parameters and is invariant under trace selection. Each permutation produces three PoW tickets whose joint success probability $q = 1 - (1 - p)^3 \approx 3p$, $p = T/2^{248}$, is input-independent under PRP. The distribution of valid tickets follows Binomial(P, q) where P is the total permutation count—invariant across trace choices. Multi-trial grinding (k distinct traces, selecting the best outcome) incurs $(k - 1)/k$ waste from discarded proofs, yielding net loss for $k \geq 2$. Header digest (h_H) selection is equivalent to nonce grinding under PRP. See Appendix B for the full proof.

A ASIC Architecture Details

A.1 ZK-Symbiotic ASIC Block Diagram

```
+-- ZK-SPoW ASIC (7nm, ~200W) ------------------------------------------+
|                                                                       |
| +-- Poseidon2 Core Array (50% die) --------------------------------+ |
| |                                                                  | |
| | [Core 0] [Core 1] [Core 2] ... [Core N-1]                        | |
| | Each core: Width-24 Poseidon2 pipeline (M31)                     | |
| | Mode: Symbiotic (STARK Merkle) or PoW (nonce), per-cycle MUX     | |
| |                                                                  | |
| | Per-Core:  [Input MUX] -> [Poseidon2 Pipeline R1..R30] ->        | |
| |            [Output Router: SRAM write / target comparator]       | |
| +------------------------------------------------------------------+ |
|                                                                       |
| +-- NTT Butterfly Unit (25% die) ----------------------------------+ |
| | M31 multiply-accumulate array                                    | |
| | STARK: polynomial LDE + FRI folding                              | |
| | PoW mode: idle (future: repurpose for other useful computation)  | |
| +------------------------------------------------------------------+ |
|                                                                       |
| +-- SRAM (20% die) ------------------------------------------------+ |
| | 32 MB on-chip, ~200 GB/s bandwidth                               | |
| | STARK: eval values, Merkle tree nodes, FRI intermediate data     | |
| | PoW mode: unused (Poseidon2 runs from registers only)            | |
| +------------------------------------------------------------------+ |
|                                                                       |
| +-- Control & I/O (5% die) ----------------------------------------+ |
| | PoW controller, STARK controller, scheduler, network interface   | |
| +------------------------------------------------------------------+ |
+-----------------------------------------------------------------------+
```

A.2 Poseidon2 Core Circuit (M31)

A.2.1 M31 Field Arithmetic

Modular multiplication:

```
a, b in F_p  where p = 2^31 - 1

c = a * b                  // 31 x 31 -> 62-bit product
c_hi = c[61:31]            // upper 31 bits
c_lo = c[30:0]             // lower 31 bits
result = c_hi + c_lo       // Mersenne reduction
if result >= p: result -= p    // final correction
```

Mersenne property: $2^{31} \equiv 1 \pmod{p}$, so the upper bits fold directly into the lower bits with a single addition.

Metric	M31	Goldilocks ($2^{64} - 2^{32} + 1$)
Multiplier width	$31 \times 31 \to 62$ bit	$64 \times 64 \to 128$ bit
Reduction	1 addition	shift + sub + add
Gate count	$\sim 1{,}000$	$\sim 3{,}500$
Latency	1 cycle	2–3 cycles
Area ratio	**1×**	3.5×

Table 15: M31 vs Goldilocks field arithmetic.

M31 advantage: $3.5\times$ more multipliers per die $\to 3.5\times$ more Poseidon2 cores $\to$ proportionally higher hashrate.

A.2.2 S-box: $x^5 \bmod p$

```
x --> [MUL] --> x^2
      [MUL] --> x^4 = (x^2)^2
      [MUL] --> x^5 = x^4 * x

Critical path: 3 sequential multiplications
Multipliers per S-box: 3 (with x^2 reuse)
Gate count (M31): 3 x 1,000 = ~3,000 gates
```

A.2.3 MDS Matrix (Poseidon2)

External rounds use a block-circulant structure based on the Stwo/HorizenLabs M4:

$$M_4 = \begin{pmatrix} 5 & 7 & 1 & 3 \\ 4 & 6 & 1 & 1 \\ 1 & 3 & 5 & 7 \\ 1 & 1 & 4 & 6 \end{pmatrix}$$

For each 4-element group:

```
s0' = 5*s0 + 7*s1 + s2 + 3*s3
s1' = 4*s0 + 6*s1 + s2 + s3
s2' = s0 + 3*s1 + 5*s2 + 7*s3
s3' = s0 + s1 + 4*s2 + 6*s3

Coefficients in {1,3,4,5,6,7} -- all shift+add, no multipliers
```

Width $24 = 6$ groups of $4 \rightarrow 6$ M4 applications $\rightarrow$ **shift+add only** in external MDS.

Internal rounds use sparse MDS: $M_I = \mathbf{1}\mathbf{1}^\top + \mathrm{diag}(V)$, where $V = [-2, 1, 2, 4, \dots, 2^{22}]$ (Plonky3 production values [10]):

```
sum = s0 + s1 + ... + s_{t-1}       // t-1 additions
s_i' = V[i] * s_i + sum             // V[i] in {powers of 2, -2} -> shift+add

Width 16 (standard): 16 multiplications per internal round
Width 24 (extended): 24 multiplications per internal round  (+50%)
```

A.2.4 Full Round vs Partial Round

Component	External round ($\times 8$)	Internal round ($\times 22$)
S-box	$24 \times 3K = 72K$ gates	$1 \times 3K = 3K$ gates
MDS	M4 blocks ($\sim 12K$ gates)	diag $+\mathbf{1}\mathbf{1}^\top$ ($\sim 24K$ gates)
Round constants	$24 \times 100 = 2.4K$	$2.4K$
Subtotal	**$\sim 86K$ gates**	**$\sim 29K$ gates**

Table 16: Per-round gate counts.

Total core (pipelined):

$$8 \times 86K + 22 \times 29K = 688K + 638K \approx 1{,}326K \text{ gates (Width 24)}$$

Standard Width 16: $8 \times 58K + 14 \times 21K = 464K + 294K \approx 758K$ gates. Overhead: $+75\%$ core logic.

A.3 Pipeline Design Options

A.3.1 Folded (Area Minimal)

```
1 round of hardware x 30 iterations

Area: ~86K gates (one external round circuit, shared)
Throughput: 1 hash / 30 cycles per core
@ 1 GHz: ~33M perm/sec per core -> 100M effective hash/sec (3 tickets)
```

A.3.2 Full Pipeline (Throughput Maximal)

```
30 stages, all rounds instantiated

Area: ~1.4M gates per core (1,326K logic + 108K registers + 3K control)
Throughput: 1 perm / cycle per core (after pipeline fill)
Latency: 30 cycles
@ 1 GHz: 1G perm/sec -> 3G effective hash/sec (3 tickets)
```

A.3.3 Partial Pipeline (Balanced)

```
k stages x (30/k) iterations

k=5:  ~300K gates, 1 perm / 6 cycles  -> ~167M perm/sec -> 500M eff
k=10: ~550K gates, 1 perm / 3 cycles  -> ~333M perm/sec -> 1G eff
k=15: ~700K gates, 1 perm / 2 cycles  -> ~500M perm/sec -> 1.5G eff
```

A.3.4 Die-Level Comparison

Assuming 60M gate die, 50% allocated to Poseidon2 cores (30M gates). Each permutation produces **3 PoW tickets**:

Pipeline style	Gates/core	Cores	Perm/sec/core	Effective hash/sec
Folded ($\times 1$)	86K	348	33M	**35G**
5-stage	300K	100	167M	**50G**
10-stage	550K	54	333M	**54G**
Full ($\times 30$)	1.4M	21	1G	**63G**

Table 17: Die-level comparison (60M gate die, 50% to Poseidon2, 3 tickets/perm).

Full pipeline achieves highest throughput. The 3-ticket multiplier makes full pipeline particularly effective.

A.4 Header Pre-absorption

Block header H is variable-length (parents, tx_root, timestamp). Hashing it every nonce attempt is wasteful.

Optimization: pre-absorb header into Poseidon2 sponge state.

```
On header change (~100 times/sec at 100 BPS):
  h_H = PoseidonSponge(parent_hashes || tx_merkle_root || timestamp)
  -> 8 M31 elements, stored in header_state register

Per nonce attempt (compression function mode, width 24):
  S[0..7]   <- v1   (nonce part 1)
  S[8..15]  <- v2   (nonce part 2)
  S[16..23] <- h_H  (from register, constant)
  -> 1 Poseidon2 permutation (30 rounds)
  -> output S[0..7], S[8..15], and S[16..23] compared against target (3 tickets)
```

Cost per nonce: exactly 1 Poseidon2 permutation. Header pre-hash is amortized across $\sim$10M nonce attempts per header change.

Hardware for pre-absorption: *header_state* register: 8×31 bits = 248 bits ($\sim$400 gates). Negligible compared to Poseidon2 core ($\sim$1.4M gates).

A.5 SRAM Bandwidth and Throughput Allocation

A.5.1 STARK Memory Access Pattern

Each Poseidon2 Merkle hash requires:

```
Read:  left_child  = 8 x 4 bytes = 32 bytes
Read:  right_child = 8 x 4 bytes = 32 bytes
Write: parent_node = 8 x 4 bytes = 32 bytes
Total: 96 bytes per hash
```

A.5.2 Throughput Calculation

$$\text{SRAM bandwidth} \approx 200 \text{ GB/s} \quad \text{(ideal; see note below)}$$
$$\text{Bytes per STARK hash} = 96$$
$$\text{STARK hash throughput} = 200\text{G}/96 \approx 2.08\text{G hash/sec}$$
$$\text{Total Poseidon2 throughput} \approx 21\text{G perm/sec (21 cores @ 1 GHz)}$$
$$\text{Effective PoW hashrate} \approx 63\text{G hash/sec (3 tickets/perm)}$$
$$\text{STARK allocation: } f_{\text{sym}} = 2.08/21 \approx 9.9\%$$

A.5.3 Interpretation

Metric	Value	Note
Hardware STARK fraction (f)	~10%	SRAM-bandwidth limited
Hardware PoW fraction	~90%	Fills idle Poseidon2 cycles
U (usefulness)	$\approx 67\%$	$t_0/t = 16/24$; width extension overhead = 33%
STARK proofs/sec	~260	2.08G/8M hashes per proof
PoW hashrate	~63G effective	21G perm/sec $\times$ 3 tickets
U_{avg}	$\approx 6.7\%$	$f \times U = 0.10 \times 0.67$

Table 18: Throughput allocation summary.

Time-averaged usefulness. $U_{\text{avg}} = f_{\text{sym}} \times U \approx 0.10 \times 0.67 \approx 6.7\%$ under 200 GB/s SRAM bandwidth with the STARK prover continuously active. This means 6.7% of all Poseidon2 cycles advance ZK proofs; the remaining 93.3% provide PoW security only. In a conventional PoW network, $U_{\text{avg}} = 0\%$: all mining energy produces security and nothing else. ZK-SPoW's 6.7% represents computation that would otherwise have no useful output beyond block validation. U_{avg} scales with memory bandwidth: ~13% at 400 GB/s, ~40% at 1.2 TB/s (§A.5.4). When the STARK prover is inactive (no ZK demand), $U_{\text{avg}} = 0\%$ and the ASIC operates as a conventional PoW miner.

f is not waste—it is a throughput allocation metric. It describes how Poseidon2 cycles are allocated between STARK (memory-bandwidth-limited) and PoW (compute-limited). $U \approx 67\%$ because 8 of 24 input state elements carry *header_digest* rather than ZK data. The PoW cycles provide additional network security as a low-marginal-cost byproduct. The two workloads are complementary: PoW is compute-bound, STARK is memory-bound. They share Poseidon2 cores but bottleneck on different resources, achieving near-perfect utilization.

SRAM bandwidth assumption. The 200 GB/s figure assumes ideal conditions. Routing overhead, bank conflicts, and NTT/Poseidon2 arbitration may reduce effective bandwidth by 10–30%, proportionally lowering f_{sym} and U_{avg}. The qualitative conclusions (complementary bottleneck, PoW-dominated allocation) are robust to this range.

Width-24 efficiency. Because STARK Merkle throughput is SRAM-bandwidth-limited at ~2.08G hash/sec, Width-24 compression (1 perm/hash) delivers the same ZK proof rate as Width-16 sponge (2 perm/hash) while consuming half the Poseidon2 cycles (~10% vs ~20% of core capacity). The freed cycles serve PoW. Higher SRAM bandwidth increases ZK proof *economic throughput* (more proofs/sec) but does not change U.

A.5.4 Increasing STARK Throughput

With HBM, the STARK fraction approaches 100%, and nearly all Poseidon2 cycles serve STARK computation simultaneously with PoW. Note: this increases ZK proof *economic throughput* but

Memory technology	Bandwidth	f	U_{avg} ($f \times 67\%$)	Proofs/sec
SRAM 32 MB	200 GB/s	~10%	~6.7%	~260
SRAM 64 MB	400 GB/s	~20%	~13%	~520
HBM3 8 GB	1.2 TB/s	~60%	~40%	~1,560
HBM3E 16 GB	2.4 TB/s	~100%	~67%	~3,120

Table 19: U_{avg} scaling with memory bandwidth.

does not change U (which is bounded by $t_0/t = 16/24 \approx 67\%$, the width extension overhead). Higher memory bandwidth cannot overcome the fundamental width ratio cost.

A.6 Die Area: kHeavyHash vs Poseidon2-PoW

A.6.1 kHeavyHash Core

```
cSHAKE256 hash (x2):       ~80K gates (2x Keccak-f[1600], ~40K each)
64x64 nibble matrix mul:   ~65K gates (integer matmul + XOR)
Control:                    ~5K gates
Total:                     ~150K gates per core
Throughput:                ~1G hashes/sec per core @ 1GHz

Note: Keccak-f[1600] gate counts range from ~12K (compact/folded) to
~120K (fully pipelined). The ~40K estimate assumes a mid-pipeline
design (4-8 round stages) balancing area and throughput for ASIC mining.
```

A.6.2 Poseidon2-PoW Core (M31, Width 24, Full Pipeline)

```
S-box circuits:            ~642K gates
  External: 24 x 3K x 8 rounds = 576K
  Internal:  1 x 3K x 22 rounds = 66K
MDS circuits:              ~624K gates
  External: 6 M4 blocks x ~2K x 8 rounds = 96K (shift+add only)
  Internal: 24 x 1K x 22 rounds = 528K
Round constant storage:    ~55K gates (214 constants x 31 bits)
Pipeline registers:        ~108K gates (24 x 31 bits x 29 stages)
Input MUX + output router: ~1K gates
PoW controller:            ~2K gates (header reg, nonce counter, triple target comparator)
Total:                     ~1.4M gates per core
Throughput:                ~1G perm/sec -> 3G effective hash/sec (3 tickets) @ 1GHz
```

A.6.3 Chip-Level Comparison

Metric	kHeavyHash ASIC	Poseidon2-PoW (Width 24)
Core area	~150K gates	~1.4M gates
Cores (60M gate die)	~380 (95% utilized)	~21 (50% allocated)
Throughput per core	~1G/s	~1G perm/s → 3G eff/s (3 tickets)
Total chip hashrate	~380G/s	~63G/s effective
ZK proof capability	None	~260 proofs/sec
Additional components	None	NTT, SRAM, controller

Table 20: Chip-level comparison: kHeavyHash vs Poseidon2-PoW.

Poseidon2 has ~6× lower PoW hashrate per die than kHeavyHash. **This is absorbed by difficulty adjustment**—all miners use the same hash function, so per-miner revenue is deter-

mined by hashrate share, not absolute hashrate. The ZK proof capability provides additional revenue unavailable to kHeavyHash miners.

A.7 M31 vs Goldilocks ASIC Comparison

Metric	Goldilocks ($2^{64} - 2^{32} + 1$)	M31 ($2^{31} - 1$)
Element size	64 bits	31 bits
Multiplier gates	~3,500	~1,000
Multiplier latency	2–3 cycles	1 cycle
Poseidon2 width (ext.)	13 (rate 9, cap 4)	24 (compression)
Hash output	$4 \times 64 = 256$ bits	$8 \times 31 = 248$ bits
Cores per die (30M gates)	~23	~21
STARK ecosystem	Plonky2/Plonky3	**Stwo (potential Kaspa choice)**

Table 21: M31 vs Goldilocks ASIC comparison.

M31 is the natural choice if Kaspa adopts Stwo. The smaller multiplier (1/3.5 area) enables higher core density and hashrate per die, while matching Stwo's field arithmetic exactly.

B Trace Grinding Analysis

We prove that trace selection in Symbiotic mode provides zero advantage for PoW mining under the PRP assumption on Poseidon2.

B.1 Assumptions

1. **PRP.** The Poseidon2 permutation $\pi : \mathbb{F}_p^{24} \to \mathbb{F}_p^{24}$ is a pseudorandom permutation. For any input x, the output $\pi(x)$ is indistinguishable from uniform over $\mathbb{F}_p^{24}$.

2. **Fixed tree sizes.** Each Merkle commitment phase i ($i = 0$ for the initial trace commitment, $i = 1, \ldots, m$ for FRI rounds) has $2N_i - 1$ internal nodes, where N_i is determined by the protocol (trace length, blowup factor, FRI folding rate). The values N_i are independent of trace content.

3. **Fixed header digest.** The header digest $h_H \in \mathbb{F}_p^8$ is fixed at the start of each Merkle commitment phase and constant throughout that phase.

B.2 Ticket Count Invariance

Each Poseidon2 permutation in the Merkle tree produces three PoW tickets. The total number of permutations across all STARK phases is:

$$P = \sum_{i=0}^{m} (2N_i - 1)$$

Since each N_i is a protocol parameter independent of the trace t:

$$\forall t_1, t_2 : \quad P(t_1) = P(t_2) = P$$

The miner cannot increase the number of PoW tickets by selecting a different trace.

B.3 Distribution Invariance

Under PRP, for any distinct inputs $x_1, \ldots, x_P$ to the permutation, the outputs $\pi(x_1), \ldots, \pi(x_P)$ are jointly pseudorandom. In a Merkle tree, all permutation inputs are distinct with overwhelming probability: two nodes sharing the same input (n_L, n_R, h_H) requires a collision in the 248-bit child outputs, which occurs with probability at most $\binom{P}{2} \cdot 2^{-248}$—negligible for $P \leq 10^7$.

Each permutation produces three PoW tickets. Under PRP, the success event for each permutation (at least one ticket below target T) is a Bernoulli trial with parameter:

$$q = 1 - (1-p)^3 \approx 3p, \quad p = T/2^{248}$$

This parameter depends only on the target T, not on the permutation input. Since inter-permutation independence holds (distinct inputs under PRP), the number of successful permutations follows:

$$V \sim \text{Binomial}(P, q)$$

Both parameters (P, q) are trace-independent. Therefore, not only the expectation $E[V] = Pq$ but the *entire distribution* of valid tickets is invariant under trace selection. There is no trace for which the variance is smaller (guaranteeing a hit) or larger.

Fiat-Shamir cascade. Trace selection affects FRI round Merkle trees via the Fiat-Shamir challenge dependency on the initial Merkle root. Changing the trace changes all subsequent challenges, folding points, and FRI Merkle trees. However, each FRI tree size N_i is protocol-determined, and PRP ensures each resulting ticket has identical success probability. The argument above extends to all commitment phases.

B.4 Header Digest Grinding

The miner can produce different header digests h_H by choosing different parent blocks or transaction sets. With the same trace but a new h_H, the entire Merkle tree must be rebuilt (cost: P permutations), producing P new permutation triples. Under PRP, this is functionally equivalent to P pure PoW nonce hashes—identical cost, identical ticket distribution. Moreover, only one h_H can be used for the STARK proof; the remaining attempts produce PoW tickets but discard the STARK computation. Header digest grinding offers no advantage beyond standard nonce grinding.

B.5 Multi-Trial Grinding

A miner who computes k distinct traces and selects the one with the most valid PoW tickets:

Cost: $k \times (C_{\text{NTT}} + C_{\text{trace}} + P)$ permutations, where C_{NTT} and C_{trace} are the NTT and trace generation costs (counted conservatively as zero in the comparison below).

Benefit: $\max(V_1, \ldots, V_k)$ where each $V_i \sim \text{Binomial}(P, q)$ independently.

For the same $k \times P$ Poseidon2 permutations in pure PoW mode, all tickets are valid (none discarded), yielding $V_{\text{PoW}} \sim \text{Binomial}(kP, q)$.

By linearity, $E[V_{\text{PoW}}] = kPq$. The best-of-k selection gives $E[\max(V_1, \ldots, V_k)] \leq Pq + O(\sqrt{\log k \cdot Pq(1-q)})$. For any $k \geq 2$:

$$E[\max(V_1, \ldots, V_k)] < kPq = E[V_{\text{PoW}}]$$

Multi-trial grinding is strictly dominated by pure PoW. Including the NTT and trace generation overhead (omitted above) makes the comparison strictly worse for grinding.

B.6 Merkle Tree Feedback Structure

In Width-24 compression, the Merkle parent output $S[0..7]$ becomes an input to the next tree level, and h_H occupies $S[16..23]$ at every level. This creates structured, non-i.i.d. inputs to successive permutations. Under PRP, the permutation's output distribution is uniform regardless of input structure. The full 30-round Poseidon2 (8 external + 22 internal) provides complete diffusion across all 24 state elements. Any weakness in PRP for structured inputs would constitute a break of Poseidon2 itself—the same assumption underlying the PoW security analysis (§6.3). □

B.7 Triple-Ticket Independence

The three PoW tickets $pow\_ticket_0 = S[0..7]$, $pow\_ticket_1 = S[8..15]$, and $pow\_ticket_2 = S[16..23]$ are outputs of the same Poseidon2 permutation and therefore deterministically linked. We show that under PRP, this linkage carries no exploitable statistical correlation.

Proposition 2. *Under the PRP assumption on Poseidon2, pow_ticket_0, pow_ticket_1, and pow_ticket_2 are mutually independent.*

Proof. Let $\pi : \mathbb{F}_p^{24} \to \mathbb{F}_p^{24}$ be a PRP. For any fixed input $x \in \mathbb{F}_p^{24}$, the output $\pi(x)$ is computationally indistinguishable from a uniform sample over $\mathbb{F}_p^{24}$.

Partition $\mathbb{F}_p^{24} = \mathbb{F}_p^8 \times \mathbb{F}_p^8 \times \mathbb{F}_p^8$ as (A, B, C) where $A = S[0..7]$, $B = S[8..15]$, $C = S[16..23]$. If (A, B, C) is uniform over $\mathbb{F}_p^{24}$, then A, B, C are mutually independent, each uniform over $\mathbb{F}_p^8$. This is a standard property of product probability spaces: the uniform distribution on a product space implies independence of coordinate projections.

Therefore:

$$P(A < T) = p = T/2^{248}$$
$$P(B < T) = p$$
$$P(C < T) = p$$
$$P(A < T \ \wedge \ B < T \ \wedge \ C < T) = p^3$$
$$P(A < T \ \vee \ B < T \ \vee \ C < T) = 1 - (1-p)^3 = 3p - 3p^2 + p^3$$

The quantity $q = 1 - (1-p)^3$ used in §6.4 and Appendix B is exact under PRP, not an approximation. □

Implication for mining. A miner who observes any one ticket gains no information about whether the other two are below target. The only way to evaluate all tickets is to compute the full Poseidon2 permutation—which already produces all three simultaneously. There is no early-termination optimization. In Symbiotic mode, $pow\_ticket_0 = merkle\_parent$: reading it for PoW comparison does not modify the value used by the STARK Merkle tree.

Distinguisher reduction. Any statistical test $\mathcal{T}$ that detects correlation among $S[0..7]$, $S[8..15]$, and $S[16..23]$ across multiple Poseidon2 evaluations can be converted into a PRP distinguisher: run $\mathcal{T}$ on π vs a truly random permutation ρ, and distinguish based on whether the test detects correlation. The advantage of $\mathcal{T}$ as a correlator equals its advantage as a PRP distinguisher. Under the PRP assumption, no such efficient $\mathcal{T}$ exists. □

References

[1] M. Ball, A. Rosen, M. Sabin, and P. N. Vasudevan, "Proofs of Useful Work," *IACR Cryptology ePrint Archive*, 2017/203, 2017. https://eprint.iacr.org/2017/203

[2] L. Grassi, D. Khovratovich, C. Rechberger, A. Roy, and M. Schofnegger, "Poseidon: A New Hash Function for Zero-Knowledge Proof Systems," in *USENIX Security Symposium*, 2021. https://eprint.iacr.org/2019/458

[3] L. Grassi, D. Khovratovich, and M. Schofnegger, "Poseidon2: A Faster Version of the Poseidon Hash Function," *IACR Cryptology ePrint Archive*, 2023/323, 2023. https://eprint.iacr.org/2023/323

[4] StarkWare, "Stwo: A STARK Prover." https://github.com/starkware-libs/stwo

[5] Y. Sompolinsky and M. Sutton, "The DAG KNIGHT Protocol: A Parameterless Generalization of Nakamoto Consensus," *IACR Cryptology ePrint Archive*, 2022/1494, 2022. https://eprint.iacr.org/2022/1494

[6] Kaspa, "kHeavyHash Specification." https://github.com/kaspanet/rusty-kaspa

[7] M. Fitzi, A. Kiayias, G. Panagiotakos, and A. Russell, "Ofelimos: Combinatorial Optimization via Proof-of-Useful-Work," in *Crypto 2022*. https://eprint.iacr.org/2021/1379

[8] I. Komargodski and O. Weinstein, "Proofs of Useful Work from Arbitrary Matrix Multiplication," *IACR Cryptology ePrint Archive*, 2025/685, 2025. https://eprint.iacr.org/2025/685

[9] Y. Bar-On, I. Komargodski, and O. Weinstein, "Proof of Work With External Utilities," arXiv:2505.21685, 2025. https://arxiv.org/abs/2505.21685

[10] Plonky3, "A Toolkit for Polynomial IOPs," `poseidon2/src/round_numbers.rs`, `mersenne-31/src/poseidon2.rs`. https://github.com/Plonky3/Plonky3 (accessed 2026-02-16).

[11] Nockchain, "The zkPoW L1." https://www.nockchain.org/ (accessed 2026-02-18).