equilibrium

QUANTUS

Proof-of-Work and Poseidon Security Review

Final Report, v1.1
22-10-2025

Equilibrium Group Oy
Meritullinkatu 1
Helsinki
Finland

TABLE OF CONTENTS

0. EXECUTIVE SUMMARY

Auditor: Equilibrium Group Oy
Client: Quantus Labs LLC

Scope: Security Review of QIP3: RSA-Shortcut Proof-of-Work and Poseidon Hash Function.

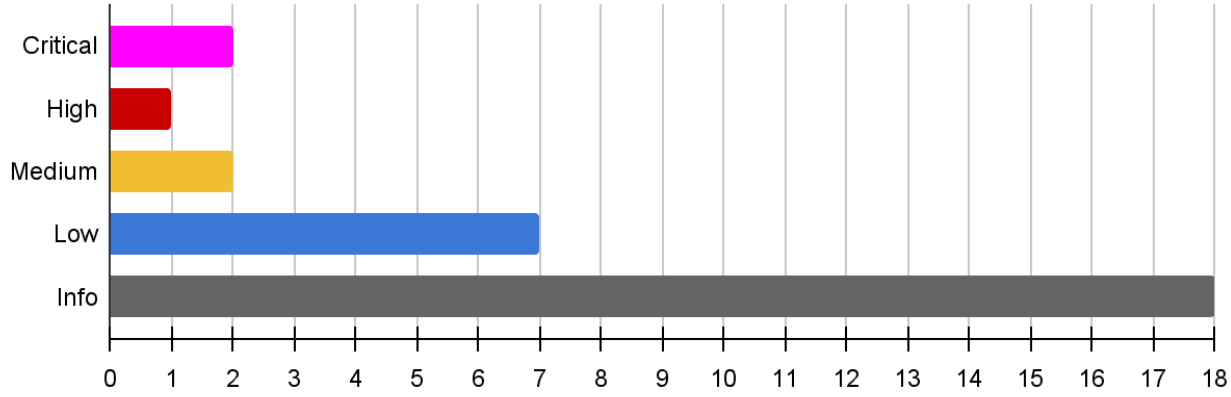
Equilibrium conducted a security review of Quantus' improvement proposal 3 (QIP3), containing a proof-of-work algorithm with a significant advantage to those able to factor 512-bit numbers. The review was conducted remotely by Equilibrium between 24-09-2025 and 22-10-2025, with an emphasis on cryptographic design soundness and implementation security/robustness.

The review covered the design and implementations of the PoW algorithm and Poseidon hash.

KEY FINDINGS:

We gathered 30 findings and observations in the review, with two marked critical in the QPoW design, namely a problem reduction inconsistency and an insufficient security parameter, that we demonstrate jointly enable quantum block generation on classical personal computers.

We identified one high severity in the use of saturating operations possibly leading to hash collisions or unexpected behavior. We identified two medium issues in hash overflow handling and encode/decode totality. Finally, we identified low or informational points relating to norm-deviations, overhead, less significant bugs, ambiguousness, documentation and structure.



We found the QIP3 and implementations well-written and well-documented modulo findings, and it was a pleasure to work with the client, providing very clear and concise descriptions and code.

1. PROJECT SUMMARY

This security review was conducted to assess the cryptographic design and soundness of the Poseidon hash function and the RSA-Shortcut Proof-of-Work algorithm. This report outlines the main findings and observations, as well as experiments and recommendations for mitigation.

The Project

The project concerns the cryptographic core of the Quantus Network. At this core is the QPoW, a quantum proof-of-work algorithm, designed to distinguish classic miners with quantum miners. The algorithm incorporates two custom hash functions, Poseidon2-256 and Poseidon2-512, derived from plonky3's poseidon2 crate. The project was selected due to its importance and wide security-impacts in a real-world deployment of the chain.

Context and Scope

The project context is the Quantus-Network (GitHub), and the project scope is

- QIP3: RSA-Shortcut: Proof of Work for Classical and Quantum Miners
  - https://github.com/Quantus-Network/improvement-proposals/blob/main/qip-0003.md
  - Commit: 0d085b8b6861dbcc06ead947fbf9d1036032face (23-09-2025)
- Poseidon Hash Function Implementation
  - https://github.com/Quantus-Network/qp-poseidon/tree/main/core
  - Commit: c39846383301538684b52b1a068887772c790df4 (24-09-2025)
- Proof-of-Work Algorithm Implementation
  - https://github.com/Quantus-Network/chain/tree/main/qpow-math
  - Commit: 13ad84f7d8b92d07fd301ebb17d3edd0c38417d4 (25-09-2025)

The goal of the assessment was to identify security vulnerabilities and correctness. We note

- Crate dependencies are out of scope,
- Additional systems, integrations or resources are out of scope,
- Tests and integration tests are out of scope,

Disclaimers

No security analysis can be a guarantee of absence of vulnerabilities. This review represents a best-effort analysis focusing on worst-case scenarios and providing directions for mitigations.

Remediation

After follow-up discussions, the client resolved all findings and major observations in:

- https://github.com/Quantus-Network/improvement-proposals/pull/17/files
  - Commit: abe1ad6fe32bb71a55d7f00d04f4cad2d73ac8f2
- https://github.com/Quantus-Network/qp-poseidon/pull/32
  - Commit: 5988016d95c66c6d358c643a3ab80a2592e12ee2

2. PROOF-OF-WORK ALGORITHM

A. FINDINGS

EQ-QNT-POW-F-01: Factoring vs RSA Inconsistency

We find QIP3 has an inconsistency on a reduction to the factoring problem vs the RSA problem. The abstract describes it *offers a significant advantage to anyone who can quickly factor 512 bit numbers*, which is consistent with the Proof-of-Work algorithm and implementation. However, the body of QIP3 and the implementation refer to creating/solving the RSA problem, which we do not find consistent with the algorithm design or implementation.

We observed that get_random_rsa will sample m=H(...), that under appropriate assumptions, one may assume to be a uniformly random 512-bit number m. However, this will not imply that m is a product of two equally large primes, that characterizes the classic hard RSA problem. It is more a get_random_factorization, which is crucially easier to solve than the RSA problem. For example, m could be divisible with 3, that we can divide out, reducing the problem size m'=m/3 with >1 bit. Generalizing the example, this means one can factor by "peeling off" easy factors, and in the end hopefully obtain the full factorization. In practice, this is what general factoring algorithms do utilizing methods such as trial division, ECM and sieves (GFNS).

A reduction to RSA is not straight-forward, as an RSA instance typically involves multiplying *two known prime numbers*, and threshold-RSA/MPC may be needed to ensure the trapdoor is unknown, which further involves committees and chain integration outside the scope. On the other hand, removing the RSA reference and keeping consistent with factoring may work, but require careful analysis or handling of easy instances, and much larger parameters, as a factoring challenge of N random bits naturally embeds an overhead in small/easy factors; that can be avoided entirely.

Recommendation
Ensure the QPoW reduces to consistent hard problem instances. We propose directions in Appendix C for an approach based on the Discrete Logarithm problem.

EQ-QNT-POW-F-02: Insufficient Fixed Security Parameter

We find the 512-bit security insufficient to separate quantum and classic abilities for the RSA problem, which is significantly worsened by the factoring problem (EQ-QNT-POW-F-01).

We base this on the RSA factoring challenge, with 512-bits factored in 1999, and up to 829 bits as of 2020 [4]. We validated our findings in the experiment section by producing quantum blocks and describing a fork attack where quantum block mining becomes easier to solve than classic mining.

Recommendation
Ensure sufficient or adjustable security level. We recommend concrete security parameters for an approach based on the discrete logarithm in Appendix C.

EQ-QNT-POW-F-04: Unexpected Behavior from Saturating Operations

The code uses undocumented saturating_add in

```
pub fn hash_to_group_bigint(h: &U512, m: &U512, n: &U512, solution: &U512) -> U512 {
    let sum = h.saturating_add(*solution);
    mod_pow(m, &sum, n)
}
```

and twice in the mining function:

```
let mut value = mod_pow(&m, &block_hash_int.saturating_add(nonce_u), &n);

...
nonce_u = nonce_u.saturating_add(U512::from(1u64));
```

This behavior is inconsistent with the specification, and can lead to critical unexpected behavior, for example giving the same hash for different blocks and/or solutions, or assigning the same nonce if starting from a large enough value. We remark saturating operators are present in dependencies and systems outside the scope, for which our recommendations also hold.

Recommendation
We recommend checked operations with graceful handling.

Alternatively, for this particular case, follow the QIP3 specification and do (big-)integer addition when adding exponents - given we work with an *unknown* order of m - and adjust `mod_pow` and `mod_pow_next` to accept larger exponents, for example an iterator over bits or words.

EQ-QNT-POW-F-03: Off-by-One in Miller-Rabin Base Selection

Deterministic test bases for the Miller-Rabin primality test are generated in is_prime as:

```
// Use the hash to generate a base between 2 and n-2
let hash = U512::from_big_endian(&poseidon_bytes);
let base = (hash % (*n - U512::from(4u32))) + U512::from(2u32);
bases[base_count] = base;
base_count += 1;
```

However, since hash % (n-4) maps to [0; n-5], adding 2 will map into [2, n-3] instead of [2, n-2]. This is off-by-one with the comment and the standard Miller-Rabin sampling interval.

Recommendation:
Subtract 3 instead of 4.

EQ-QNT-POW-F-05: Undocumented Black-Box Reductions

Recently the SHA-family was replaced with the Poseidon family of hash functions in QPoW, and notably, Poseidon works over prime fields. When the QPoW algorithm applies the XOR operation on four field elements, a gap is introduced between the max-value of u64 and the modulus p. This is then directly used as the distance function against the mining target difficulty. When assuming the hash function returns uniformly random bits (black-box assumption), this is fine, but in reality, the domains do not match up, and the reduction (gap) is not documented.

Recommendation: Document gaps/assumptions made.

B. OBSERVATIONS

EQ-QNT-POW-O-01: Unnecessary Round Constant Derivations

The Poseidon hash function is instantiated by the below functions via `Poseidon2Core::new()`

1. `get_random_rsa`
2. `hash_to_group_bigint_poseidon`
3. `mine_range`
4. `is_prime`

This instantiation recomputes the hash function round constants. Checking if a single block is valid therefore requires generating the Poseidon round constants four times; zero are needed.

Recommendation
Ensure a single generation only. See also EQ-QNT-HASH-O-01.

EQ-QNT-POW-O-02: Undocumented Unbounded Function

We remark that `get_random_rsa` has an infinite rejection sampling loop that theoretically can run unbounded. Although the expected number is reasonable in practice, it is not documented.

Recommendation
Document the expected number of iterations by analysis or measurement, or bound the retries.

EQ-QNT-POW-O-03: Ambiguous Security Claim for Primality Testing

The Miller-Rabin primality test given 32 rounds claims a security bound of 4^-32=2^-64. This requires the test bases to be selected at truly random. But **`is_prime`** (line 271-290) samples pseudo-randomly with some security parameter that would normally also appear in this bound, or more generally, the primality test security is now computational rather than statistical.

Recommendation
Clarify the bound is from standard Miller Rabin assuming truly random bases/black box.

EQ-QNT-POW-O-04: Code Structure and Left-over Comments

We observed is_prime is after the test module, while it is the norm to keep all code before tests. We noticed some comments could be removed, for example “no split chunks by Nik” and a commented out call to log::info.

3. POSEIDON HASH FUNCTION

A. FINDINGS

EQ-QNT-HASH-F-01: Undocumented Seed Constant

The Poseidon seed constant is defined as:

```
/// Fixed seed for deterministic constant generation
const POSEIDON2_SEED: u64 = 0x189189189189189;
```

We find the choice of seed constant lacks documentation. We remark, we find it unlikely that this seed, in combination with the random number generator, would produce Poseidon round constants providing significant advantages for attacks.

Recommendation: Show nothing-up-my-sleeve by choosing a common constant e.g. 0, 1, π.

EQ-QNT-HASH-F-02: Weak RNG for Round Constants

Round constants for Poseidon are recommended to be generated by Grain LFSR [5] in self-shrinking mode, see also material A and E in the original paper [1]. This brings benefits such as published round constants are commonly reproducible, deterministic and determined by the choice of Poseidon parameters. However, designers often choose their own primitives and random number generators for uniform sampling.

The Poseidon hash function uses the following code to instantiate the external permutation:

```
pub fn new() -> Self {
    let mut rng = ChaCha8Rng::seed_from_u64(POSEIDON2_SEED);
    let poseidon2 = Poseidon2Goldilocks::<12>::new_from_rng_128(&mut rng);
    Self { poseidon2 }
}
pub fn with_seed(seed: u64) -> Self {
    let mut rng = ChaCha8Rng::seed_from_u64(seed);
    let poseidon2 = Poseidon2Goldilocks::<12>::new_from_rng_128(&mut rng);
    Self { poseidon2 }
}
```

The RNG is used by the crate p3-poseidon2 [2] to sample round constants, see Appendix B.
We note the choice of ChaCha8 as RNG could be strengthened by choosing ChaCha20, or a RNG more widely accepted as cryptographically secure. While some argue 8 rounds is sufficient for ChaCha [3], this is not the commonly accepted security level for cryptographic security, and it is

generally used as a (weak) randomness generator. For example, Golang uses ChaCha8 for randomness but not for cryptographic randomness. One could argue cryptographic security is not necessary for Poseidon round constant generation, but it will inevitably make proofs weaker.

For the specific application of generating (many) round constants for the Poseidon hash, we remark it unlikely that this seed and the ChaCha8-cipher would provide any significant attacks.

Recommendations
We propose strengthening round constant generation with Grain LFSR [1] or ChaCha20/AES.

EQ-QNT-HASH-F-03: Redundant Hashing in Hash512 (and RandomRSA)

Observe the `hash512` function given as:

```
pub fn hash_512(&self, x: &[u8]) -> [u8; 64] {
    let first_hash = self.hash_padded(x);
    let second_hash = self.hash_padded(&first_hash);

    let mut result = [0u8; 64];
    result[0..32].copy_from_slice(&first_hash);
    result[32..64].copy_from_slice(&second_hash);
    result
}
```

Padding and encoding aside, the implementation implies at least 2x190 invocations of the permutation function given the two separate calls to hash_padded. Unless external applications require it, only one call with ~191 permutation calls is needed, given the sponge construction was designed for variable output lengths originally, see also [1]. Similarly, **`get_random_rsa`** in QPoW could also sample (m,n) from a single sponge, saving a further 190 for this call.

Recommendations
Extract 512 bits of hash output directly from a single sponge invocation. This can be done by modifying the end of `hash_no_pad` to be inspired by the following variant producing 2x256-bits digest, only invoking the permutation one more time and extracting more output bits.

```
let h1 = &state[..RATE];
self.poseidon2.permute_mut(&mut state);
let h2 = &state[..RATE];
(digest_felts_to_bytes(h1), digest_felts_to_bytes(h2))
```

EQ-QNT-HASH-F-04: Redundant Field Padding and Permutation

The function `hash_no_pad` has multiple levels of padding. Recall rate=r=4.

- The last input chunk C will be padded with a '1' in the main loop unless |C| % r == 0:

```
if j == num_chunks - 1 {
    if chunk.len() < RATE {
        block[chunk.len()] = Goldilocks::ONE;
    } else { unpadded = true;  }
}
```

- For the other case, the message is suffixed [1, 0, 0, 0], **but not for empty inputs**:

```
if unpadded {
    let padding_block =
        [Goldilocks::ONE, Goldilocks::ZERO, Goldilocks::ZERO, Goldilocks::ZERO];
    for i in 0..RATE {
        state[i] += padding_block[i];
    }
    self.poseidon2.permute_mut(&mut state);
}
```

- Finally, a fixed block [0, 0, 0, 1] is *always* appended

```
let padding_block = [Goldilocks::ZERO, Goldilocks::ZERO, Goldilocks::ZERO,
Goldilocks::ONE];
for j in 0..RATE {
    state[j] += padding_block[j];
}
self.poseidon2.permute_mut(&mut state);
```

We observe the first two padding rules, if removing the "but not for empty inputs" logic, jointly form the *Variable-Input-Length Hashing* originally proposed in Poseidon paper [1], namely add a '1' after the message, then fill with zeros up to the given rate. Hence, unless required by external applications, the final fixed padding block [0, 0, 0, 1] is redundant. For the special case of an empty input, as rationaled in the comment for this code block, the Poseidon construction imply a single [1, 0, 0, 0] block is hashed, which will be naturally covered with the two first padding rules once the "but not for empty input" logic is removed.

Recommendations

We recommend changing the padding scheme to the variable-input-length padding [1].

EQ-QNT-HASH-F-05: Ignoring Overflows in Digest Conversion

For the function `injective_bytes_to_felts` and `digest_bytes_to_felts`, a finite field value is created using an external call, ignoring if an overflow happened. For the first function, this is safe given only 4 bytes are encoded in a 64-bit prime. For the second function, this is only safe assuming an adversary cannot control the inputs, since it is easy to produce a hash function collision otherwise, that could lead to critical security exploitation. For example, one can use "0" as a value for one 64-bit block and "p" for another, that will provide the same Poseidon hash, as the values are equal modulo p.

Verifying all external usage of the function is out of scope, but we remark, we have only observed it used externally (out of scope) on uncontrollable digests.

Recommendations
When mapping 8 bytes into a field element, check for overflows and handle it gracefully.

Alternatively, mark the function as unsafe and document the requirement of uncontrollability, and when used, mark the invocation with a comment on why the requirement is satisfied.

For the function `injective_bytes_to_felts`, add a comment above from_int(..) explaining why this case is indeed safe; or debug_assert for no overflows, also guarding against endianness-bugs.

EQ-QNT-HASH-F-06: Undocumented (No-)Domain Separation

The core function `hash_no_pad` accepting field elements can produce the same hash as `hash_no_pad_bytes` accepting bytes. This may not be ideal if an external application expects the domains to be separated.

Recommendation: See EQ-QNT-HASH-O-04 for a proposal using type-safety.

EQ-QNT-HASH-F-07: Inconsistent Encoding and Decoding

The `injective_felts_to_bytes` have:

```
match marker_index {
    Some(j) => bytes.extend_from_slice(&last[..j]),
    None => bytes.extend_from_slice(last), // graceful fallback
}
```

This graceful fallback makes it inconsistent with the encoding, which is not documented. This could cause security issues if the external context assumes a total function.

Recommendation:
Remove the fallback. Alternatively, document or add optional skip-marker-flag to the encoding

B. OBSERVATIONS

EQ-QNT-HASH-O-01: Repeated Generation of Round Constants

The Poseidon2Core structure is instantiated when used e.g. by PoW/Wormholes, hence round constants are computed repeatedly. Depending on external usage, this should be avoided.

Recommendation: Compute only once (/once per core) on startup (/first invocation). This will also be beneficial if choosing a heavier RNG for the round constants, see EQ-QNT-HASH-F-02.

EQ-QNT-HASH-O-02: Ambiguous Hash512 Function Name

Classically the security level is suffixed hash-functions, e.g. SHA-512, but Hash512 only refers to the digest size and not the security level, that would require different sponge parameters.

Recommendation: Rename the function e.g. hash2, or see EQ-QNT-HASH-O-04.

EQ-QNT-HASH-O-03: Inefficient Byte Packing

The byte packing function `injective_bytes_to_felts` pack 4 bytes into each Goldilocks element. This drops the information rate to roughly 50%, that could be avoided.

Recommendation: For each element, pack in 7 bytes being 58 bits, alternatively up to 63 bits.

EQ-QNT-HASH-O-04: Unnecessary Memory Shift during Hashing

Inside `hash_padded_felts` is a memory shift via vector insertion before computing the hash:

```
x.insert(0, Goldilocks::from_int(len));
```

The insertion operation has O(len) time-complexity and is inefficient for large inputs.

Proposal: We propose introducing a new structure, for example called Poseidon2State and replacing Poseidon2Core, with a signature (alternatively a trait and implementations) based on:

- `fn init() -> Self`
- `fn append(&mut self, blocks: &[Goldilocks])`
- `fn finalize(self) -> [u8; 32]`

This follows standard hash design/traits and supports when the length is not known up-front by accumulation, and enables handling padding gracefully in the finalization function.

This core hasher can be exposed externally directly, or be used inside variants. For example, one could introduce a "ByteHasher", "StringHasher", "LengthPrefixedHasher", "Minimum160-BlockHasher" and/or combinations there-of in a "CircuitPaddingHasher".

This restructuring would avoid the memory shifting entirely by appending the length initially.

EQ-QNT-HASH-O-05: Notes on Code Structure

The implementation consists of a single lib.rs file at a reasonable 565 lines, which we remark is well-structured and -commented. Restructuring could possibly benefit readability and modularity.

Proposal:
We propose moving helper functions, serialization and associated tests to serialization.rs.

1. `injective_string_to_felts`
2. `injective_felts_to_bytes`
3. `digest_felts_to_bytes`
4. `digest_bytes_to_felts`
5. `injective_bytes_to_felts`
6. `u128_to_felt`

We propose introducing a core.rs with a Poseidon2State per EQ-QNT-POS-O-04.

We propose a file for each hasher, or for simplicity, only one for the circuit-padding-variant.

We propose rewriting lib.rs to focus on no-std and re-exporting the public interface only.

Finally, we remark one could use a common hash trait for interchangeability.

EQ-QNT-HASH-O-06: Wrong Comment on XOR vs Field Addition

In the Poseidon main loop, a comment says *"XOR with state prefix (chaining)"*, but this is prime field addition and not the bitwise exclusive-or operation.

Recommendation: Change to e.g. "Add chunk to state"

EQ-QNT-HASH-O-07: Ambiguous Padding in Public API

The public API offers `hash_padded_felts` and `hash_no_pad(_felts)`, but multiple paddings are applied, which could be confusing from a public API point-of-view. In particular, the two above functions are different in the sense that *both* pad with the `1` suffix and the fixed [0, 0, 0, 1] block, but the former further left-pad a length prefix and right-pad a minimum width.

Recommendation: Rename the special case to circuit padding, or see EQ-QNT-HASH-O-04.

4. OTHER OBSERVATIONS

EQ-QNT-MISC-O-01: Not Constant-Time

We recommend marking in the README that crypto-implementations are variable-time.

EQ-QNT-MISC-O-02: Left-Over Comments

We note both projects have left-over comments e.g. TODOs for missing tests.

EQ-QNT-MISC-O-03: Note on High-level Documentation

We find the sequential code and steps well-commented, however remark the implementation would benefit from more high-level documentation per function and per module. As an example,

```
/// Convert field elements back to bytes in an injective manner
pub fn injective_felts_to_bytes(input: &[Goldilocks]) -> Vec<u8> { .. }
```

could naturally describe the injective map/padding scheme.

EQ-QNT-MISC-O-04: Documentation on Security Reductions

We find parts of the code could be more precise on the security claims, or point to external documentation, see also EQ-QNT-POW-F-05. Beside this a comment could be ambiguous:

```
// => 256 bits of classical preimage security => 128 bits of quantum preimage security
```

Namely it assumes Grover's search is the best possible attack, which is not documented or generally true. For example, an RSA-based hash function could be at risk for quantum security.

EQ-QNT-MISC-O-05: Undocumented Mixed Endianness

We observed the PoW-algorithm uses big-endian (U512), while Poseidon uses little-endian for byte-packing. We propose documenting this, or staying consistent with a single endianness.

EQ-QNT-MISC-O-06: Similar Primitives across Crates

We observed e.g. `digest_bytes_to_felts` exists in both

- `qp-zk-circuits/common/src/utils.rs` on the stack (out of scope), and
- `qp-poseidon/src/lib.rs` on the heap,

which may be fine for different purposes, but could be centralized in a common repository.

EQ-QNT-MISC-O-07: Note on Exact Chunk Iteration

We observed multiple iterations with index management handling the last block, for example in `injective_bytes_to_felts`, where Rust's `.chunks_exact()` and `.remainder()` are useful and more robust against off-by-ones.

EQ-QNT-MISC-O-08: Tests, Fuzzing and Integration Tests

We remark the Poseidon hash crate contains 17 tests while the QPoW have two. Further testing could improve the robustness of the implementations, especially test-vectors and edge-cases.

5. EXPERIMENTS AND FORK ATTACK

This section shows experiments and a practical attack on the QPoW following the findings.

We first ran the chain in development mode to generate a real block hash, then computed (m,n):

```
let block_hash = hex!("57637ad86180b4dd103c53b271de6d32ef8a70c8e42b504f176de9d354a8cb27");
let (m, n) = get_random_rsa(&block_hash);
println!("m={}", m);
println!("n={}", n);
```

This results in

```
m=625572107497236201373429353623221167375696521658809166413927884673164851856 38
n=1149006150640652375627521917205356853437729331606583471563841523887728775470 3514
591589069679748457691076698204379744877023842686229250279726829860536804333
```

We ran YAFU (Yet Another Factoring Utility) [6] on a PC with 9950X3D CPU and 64GB RAM. YAFU is an automated integer factorization stack combining other free software factorization algorithms like ECM and GFNS, and is optimized for general inputs up to 160 digits ~ 532 bits.

```
yafu-x64
factor(1149006150640652375627521917205356853437729331606583471563841523887728775470 3
51459158906967974845769107669820437974487702384268622925027972682986053680 4333)
-threads 32
```

This produced the factorization in 19 seconds relying on ECM:

```
P4 = 2161
P13 = 1549924181159
P32 = 61774065769472540638468707495173
P107 =
55532975826427260612251374447041038413165132723468232185671857444360006426670280 80
75344192473346367695724 79
```

We will target zero distance (quantum miner), hence we look for non-zero s for the equation:

$$m^{h+s} \equiv m^{h+0} \quad \Leftrightarrow \quad m^h m^s \equiv m^h \quad \Leftrightarrow \quad m^s \equiv 1 \pmod{n}$$

Any multiple of the order of m mod n works as a choice of **s**, that can be computed given the prime factorization. For simplicity and given this example with square-free n, we compute s as the square-free Carmichael function, that is always a multiple of the order of m mod n:

$$\lambda(n) = lcm(P4 - 1, P13 - 1, P32 - 1, P107 - 1) = s$$

Hence a valid quantum solution to the block above is:

```
s=23926551031372511350460143598781778045479196788736623405120994994467562966168242
826007854599915285040216628948716291700118243092891466097534864639121576 0
```

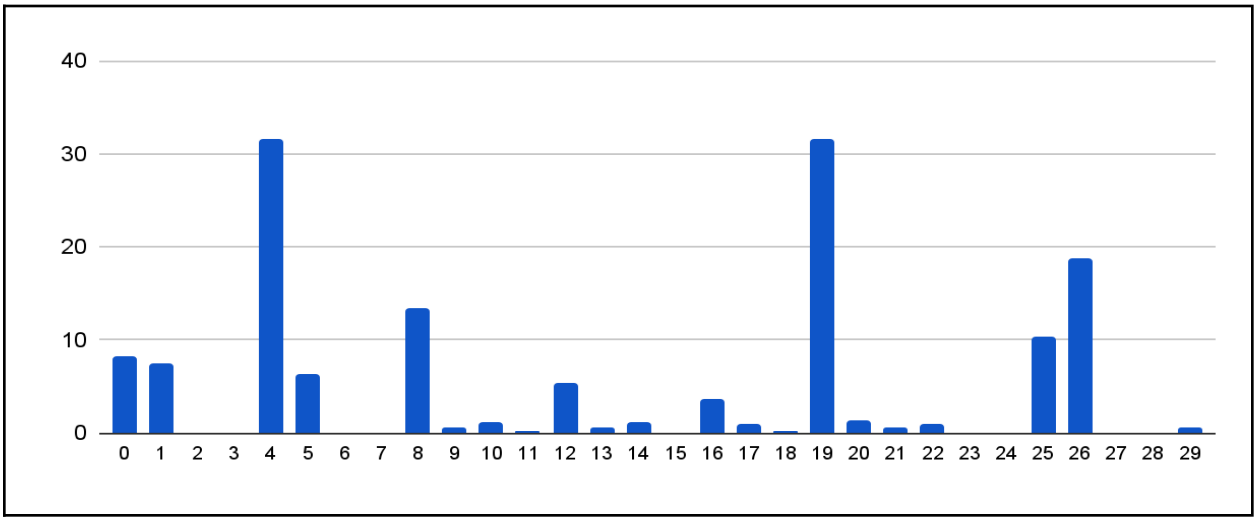
We extend `qpow-math/src/lib.rs` with a test:

```
#[test]
fn test_is_valid_shortcut() {
    // Blockhash h
    let block_hash = {
        let mut arr = [0u8; 32];
        hex::decode_to_slice(
            "57637ad86180b4dd103c53b271de6d32ef8a70c8e42b504f176de9d354a8cb27",
            &mut arr,
        ).unwrap();
        arr
    };

    // Nonce s
    let nonce_bytes = {
        let bytes =
BigUint::from_str("23926551031372511350460143598781778045479196788736623405120994994467562966168242826007854599915285040216628948716291700118243092891466097534864639121576 0")
            .unwrap()
            .to_bytes_be();
        let mut arr = [0u8; 64];
        arr[64 - bytes.len()..].copy_from_slice(&bytes);
        arr
    };

    // Verify trapdoor
    let threshold = U512::from(0u64);
    let (ok, value) = is_valid_nonce(block_hash, nonce_bytes, threshold);
    println!("is_valid_nonce = {}, distance_achieved = {}", ok, value);
}
```

Producing a successful validation and attack:

```
is_valid_nonce = true, distance_achieved = 0
```

To verify the blockhash was not a unique case, we simulated random blockhashes by factoring SHA512(x) for nonces x=0,..,29. We measured the seconds to factor, where empty points represent timeouts when factoring exceeded 60s. See Appendix D for the table of values.



The table highlights EQ-QNT-POW-F-01, namely that some factoring instances are easy while others are hard, whereas RSA instances (subset of factoring) characterize particularly hard cases.

The experiments show that in many cases, a single personal computer can invoke the quantum trapdoor in less than a second. This critically allows for a fork attack to take control over block generation and liveness. We remark the chain is out of scope and the below attack was not tested.

Quantum Fork Attack (sketch)
In this attack, the adversary reads the latest block, and attempts to factor it “fast” e.g. within 5 seconds, or waits for the next block. Once he finds a block to factor fast, that given above and 12s blocktime is expected in minutes, he factors m and computes s as to generate a quantum block.

However the adversary could be unlucky obtaining a ”slow” (more than 5s) factoring instance after his first quantum block. However, since he is the miner, he can control the next mining instance via a mining nonce in the block header mining, or by computing *different puzzle solutions using multiples of the nonce s*, and accepts the first he can factor. At this point, the adversary has a local fork of two quantum blocks, and can continue this rejecting sampling nonces to produce a long local quantum fork - that the main chain is forced to follow - breaking issuance/mining/fairness.

Furthermore, an adversary could tamper with timestamps in his local fork to force the difficulty target to decrease the maximum 10% allowed each block. Over time the target value of mining will become so low, the adversary can effectively disable classic mining and break liveness.

6. CLASSIFICATIONS

We use standard CVSS severity classifications:

- **Critical**
- **High**
- **Medium**
- **Low**
- **None** (Information)

Generally, we mark findings **Critical** if an exploit or bug may cause a complete system compromise, outage, data loss or regulatory impact. This could be through key recovery, signature forgery or other protocol insecurities. The issues should be fixed with highest priority.

We mark findings as **High** if the exploit may have a significant security impact in degradation of service or exposure of sensitive information, or if the system is volatile under specific mis-use. This covers attacks under realistic assumptions and resource bounds that must be fixed.

Findings marked as **Medium** causes limited impact, for example minor data exposure or isolation to non-critical components. This also covers public APIs susceptible to mis-use.

The **Low** findings cause low or negligible security impact. This covers theoretical or impractical conditions, but highlights weaknesses such as weak entropy, undocumented skews in randomness, or deviation from best practices.

The **Informative** findings contain no vulnerabilities and cover general proposals and comments.

7. CONCLUSION

A security review was conducted accessing the cryptographic design and soundness of the Quantum Proof-of-Work and Poseidon Hash Function by Quantus.

The security team at Equilibrium identified a total of 30 findings and observations.

In order to mitigate vulnerabilities, we recommend minimum mitigations:

1. Correct the QPoW to ensure hard puzzles providing >=128 bits of computational security.
2. Guard the system against possible collisions happening via saturation/overflows.

We thank Quantus for their trust and look forward to future collaboration.

8. REFERENCES

1. Poseidon https://eprint.iacr.org/2019/458.pdf
2. p3-poseidon2 v0.3.0 https://crates.io/crates/p3-poseidon2
3. Too Much Crypto https://eprint.iacr.org/2019/1492.pdf
4. https://en.wikipedia.org/wiki/RSA_Factoring_Challenge (29-09-2025)
5. Martin Hell, Thomas Johansson, Alexander Maximov,and Willi Meier. The Grain Family of Stream Ciphers. In The eSTREAM Finalists, volume 4986 of LNCS, pages 179–190. Springer, 2008.
6. YAFU (Yet Another Factoring Utility) https://github.com/bbuhrow/yafu
7. Boneh, D. and Venkatesan, R. (1996). Hardness of Computing the Most Significant Bits of Secret Keys in Diffie-Hellman and Related Schemes. In *CRYPTO '96*, Lecture Notes in Computer Science, vol. 1109, pp. 129–142

9. APPENDIX A: Poseidon Permutation Parameters

	Value	Reference
Field	Prime $p = 2^{64} - 2^{32} + 1$	[1]
Width	12	[1]
Rate	4	[1]
Capacity	8	Capacity=width-rate
Sbox-degree	7	[2]
Full rounds	8	[3]
Partial rounds	22	[3]

The Sbox is invertible as intended:

$$gcd(7, p - 1) = 1$$

Given the sponge security scales with p^(c/2) assuming an ideal random permutation [4], plugging in the 64-bit Goldilocks prime gives 256 bits of classical security as intended:

$$t \leq log_2 [(2^{64} - 2^{32} + 1)^{8/2}] = 255.999...$$

References (Appendix A only):

1. https://github.com/Quantus-Network/qp-poseidon/blob/main/core/src/lib.rs
2. https://github.com/0xPolygonZero/Plonky3/blob/main/goldilocks/src/poseidon2.rs
3. https://github.com/0xPolygonZero/Plonky3/blob/main/poseidon2/src/round_numbers.rs
4. Guido Bertoni, Joan Daemen, Michaël Peeters, and Gilles Van Assche. On the Indifferentiability of the Sponge Construction. In EUROCRYPT 2008

10. APPENDIX B: StandardUniform in p3-poseidon2

https://github.com/0xPolygonZero/Plonky3/blob/main/poseidon2/src/lib.rs#L50

```
/// Create a new Poseidon2 configuration with random parameters.
pub fn new_from_rng<R: Rng>(rounds_f: usize, rounds_p: usize, rng: &mut R) -> Self
where
    StandardUniform: Distribution<F> + Distribution<[F; WIDTH]>,
{
    let external_constants = ExternalLayerConstants::new_from_rng(rounds_f, rng);
    let internal_constants = rng.sample_iter(StandardUniform).take(rounds_p).collect();

    Self::new(external_constants, internal_constants)
}
```

https://github.com/0xPolygonZero/Plonky3/blob/main/poseidon2/src/external.rs#L199

```
pub fn new_from_rng<R: Rng>(external_round_number: usize, rng: &mut R) -> Self
where
    StandardUniform: Distribution<[T; WIDTH]>,
{
    let half_f = external_round_number / 2;
    assert_eq!(
        2 * half_f,
        external_round_number,
        "The total number of external rounds should be even"
    );
    let initial_constants = rng.sample_iter(StandardUniform).take(half_f).collect();
    let terminal_constants = rng.sample_iter(StandardUniform).take(half_f).collect();

    Self::new(initial_constants, terminal_constants)
}
```

https://github.com/0xPolygonZero/Plonky3/blob/main/goldilocks/src/goldilocks.rs#L162

```
impl Distribution<Goldilocks> for StandardUniform {
    fn sample<R: Rng + ?Sized>(&self, rng: &mut R) -> Goldilocks {
        loop {
            let next_u64 = rng.next_u64();
            let is_canonical = next_u64 < Goldilocks::ORDER_U64;
            if is_canonical {
                return Goldilocks::new(next_u64);
            }
        }
    }
}
```

The above implementation excerpt documents the standard rejection sampling into [0, p-1], which ensures (safe) uniform samples in the prime field assuming access to a secure RNG.

11. APPENDIX C: Design-directions for a DL-QPoW

We remark the below serves as explorative directions, without proofs or guarantees.
We propose directions to base upon variants of the discrete logarithm problem rather than RSA or factoring. The discrete logarithm is standard to assume hard in the classical setting, yet can also be solved in polynomial time by Shor's Order Finding Algorithm in the quantum setting.

The idea is, given a block-header-digest x, we define the challenge c=H(x) where H is a random oracle/hash mapping digest x to a group in which the discrete logarithm is hard. The challenge is now for a miner to find a (DL) solution **s** such that $g^s = c$ for a fixed public generator g.

A classical miner cannot compute s given the challenge c, as the problem is assumed classical hard, but he could try mine for a nonce w.r.t. a distance function. But, a quantum algorithm exists to compute s in poly-time, hence theoretically invoke the trapdoor, that **must be handled** by the chain and external system. We remark a classical miner can resort to standard attacks like baby-step giant-step or Pollard's rho, however still insufficient for sufficient security parameters. We also remark that the DL-problem alone does not fully capture mining, which is better modelled by "approximate-DL", for which a security analysis/reduction is needed, see e.g. [7].

In the *integer* discrete log setting, we can choose a random prime q for the DL-hard subgroup, yet compute modulo safe prime p=2q+1. We choose a generator g and verify it has order q, hence generating the full DL-hard subgroup. Here H(x) could be instantiated with a hash function, and mapped appropriately to integers mod p via rejection sampling or analysis of negligible skew.

In the *elliptic curve* settings, we propose using a standard curve in which the discrete logarithm is assumed hard, for example secp256r1 (P256) or secp256k1 (Bitcoin). Care must be taken when instantiating H, for example use a proper hash-to-curve function, or rejection sampling.

We remark that if efficient zk-proofs are crucial, for example integration with the plonky3 proof system natively working over a specific 64-bit prime ("Goldilocks"), it would be very beneficial to design the QPoW algorithm for an elliptic curve over an extension field of Goldilocks for efficient proofs; however such curves are not battle-tested in the same manner as standardized ones.

For security parameters, we recommend based on NIST SP 800-56A and NIST SP 800-57:

	128 bits of security	256 bits of security
Integer Discrete Log	3072 bits	15360 bits
Elliptic Curve Discrete Log	256 bits	512 bits

12. APPENDIX D: Table of SHA512 Factoring Times

YIFU factoring times for SHA512(x) on 9950X3D CPU with 64GB RAM.

x	Seconds
0	8.3211813
1	7.4722332
2	60s timeout
3	60s timeout
4	31.6652426
5	6.2504776
6	60s timeout
7	60s timeout
8	13.4084632
9	0.6052454
10	1.2209905
11	0.2398184
12	5.3369234
13	0.5669508
14	1.0899028
15	60s timeout
16	3.707001
17	0.8828479
18	0.2381312
19	31.5098393
20	1.2799914
21	0.5780959
22	1.0102876
23	60s timeout
24	60s timeout
25	10.4009145
26	18.8304477
27	60s timeout
28	60s timeout
29	0.5861804