# Coda Cryptographic Review

## O1 Labs

May 11, 2020 – Version 1.1


### Prepared for

Brandon Kase
Izaak Meckler
Emre Tekisalp

### Prepared by

Mason Hemmel
Ava Howell
Thomas Pornin
Javed Samuel
Eric Schorn




# Executive Summary

## Synopsis

During the spring of 2020, O(1) Labs engaged NCC Group to conduct a cryptographic assessment of Coda Protocol. This cryptocurrency leverages state-of-the-art cryptographic constructions to provide traditional cryptocurrency applications with a more lightweight blockchain. This assessment focused on the core cryptographic primitives as well as the overlaid protocol.

The O(1) Labs team provided source code, and was communicative throughout the engagement. NCC Group performed this test on a purpose-built branch of the code available at the time of testing.

## Scope

NCC Group's evaluation included:

- **Compilation of Snarky to Rank-1 Constraint System (R1CS)**: This covers the process of turning the arithmetic circuits into an instantiation of an NP-complete problem for which efficient argument systems have been designed.
- **Implementations of Snarky Primitives**: The primitives implemented are a verifiable random function (VRF), Schnorr signatures, a random oracle function, and elliptic curve arithmetic in support of the above.
- **Elliptic Curves and Generators:** This includes the choice of elliptic curves, as well as the soundness of their instantiation and implementation.
- **Ledger Hardware Wallet App:** This is an implementation of the Schnorr signature scheme used to sign Coda Protocol transactions on the Ledger BOLOS platform.
- **Blockchain SNARK:** The Coda Protocol includes both a pairing-based and discrete-log-based implementation of the MARLIN preprocessing SNARK. These allow for proofs of correctness of the overall state of the blockchain at a given moment, as well as the correctness of transitions between states (i.e. blocks).
- **Overall Protocol Review:** The implementation of the Coda Protocol's core consensus mechanism (Ouroboros Samasika), ensuring that the implementation accords with the whitepaper's design, and ensuring that the parameter choices made are appropriate and do not introduce any issues in and of themselves.

Security issues arising from side-channels and issues in the upstream papers that were implemented in this project, including those authored by O(1) Labs, were not in scope for this assessment.

More detailed discussion of NCC Group's strategy in evaluating these components can be found in Appendix B on page 18.

## Key Findings

The assessment uncovered a set of common cryptographic flaws. Two of the more notable were:

- Potential mishandling of point addition edge cases, which could result in provers being forced to create invalid proofs.
- In computers that used a legacy C++ standard library or could not access cryptographically-secure pseudorandom number generators, Schnorr secret key values are generated such that attackers with knowledge of the underlying system would likely be able to predict the secret.

O(1) Labs addressed four of the six findings reported by NCC Group, including both of the above, prior to the release of this report.

## Strategic Recommendations

O(1) Labs's code, while largely stable on its own, relies in some parts on upstream libraries not covered in this audit. Time permitting, these dependencies should be audited and/or reduced to ensure the lowest possible attack surface.

# Dashboard

## Target Metadata

| | |
|---|---|
| **Name** | Coda |
| **Type** | Cryptocurrency |
| **Platforms** | OCaml, Rust |
| **Environment** | Local Instance |

## Engagement Data

| | |
|---|---|
| **Type** | Cryptographic Assessment |
| **Method** | Code-Assisted |
| **Dates** | 2019-12-23 to 2020-04-09 |
| **Consultants** | 4 |
| **Level of Effort** | 45 days |

## Targets

Blockchain SNARK Implementation

Coda Protocol Consensus Implementation

Discrete-Log-Based Marlin SNARK Implementation

Elliptic Curve Implementation in R1CS-Generating OCAML Layer

Elliptic Curve Implementation Used to Instantiate Marlin SNARK

Pairing-Based Marlin SNARK Implementation

Poseidon Hash Implementation in R1CS-Generating OCAML Layer

Random Oracle

Schnorr Signature Implementation in R1CS-Generating OCAML Layer

Verifiable Random Oracle

## Finding Breakdown

| | |
|---|---|
| Critical issues | 0 |
| High issues | 0 |
| Medium issues | 1 |
| Low issues | 2 |
| Informational issues | 3 |
| **Total issues** | **6** |

## Category Breakdown

| | |
|---|---|
| Cryptography | 5 |
| Other | 1 |

## Component Breakdown

| | |
|---|---|
| Ledger App | 2 |
| Zexe | 1 |
| camlsnark_c | 1 |
| snarky | 2 |

## Key

Critical High Medium Low Informational

# Table of Findings

For each finding, NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. For an explanation of NCC Group's risk rating and finding categorization, see Appendix A on page 16.

| Title | Status | ID | Risk |
|---|---|---|---|
| Schnorr Secret Key Values Are Predictably Generated in Legacy Architectures or With Legacy C++ Standard Library Implementations | Fixed | 004 | Medium |
| Potential Mishandling of Point Addition Edge Cases | Fixed | 001 | Low |
| Out-of-Bounds Memory Write | Fixed | 006 | Low |
| Poseidon S-Box is Not a Bijection Under MNT Curves' Field | Fixed | 002 | Informational |
| Future Developments May Impact Subgroup Security of Discrete-Log-Based Elliptic Curve | Reported | 007 | Informational |
| Ledger Signing Does Not Abort When `k = 0` | Reported | 009 | Informational |

| | |
|---|---|
| **Finding** | **Schnorr Secret Key Values Are Predictably Generated in Legacy Architectures or With Legacy C++ Standard Library Implementations** |
| Risk | **Medium** Impact: High, Exploitability: Low |
| Identifier | NCC-O1LB001-004 |
| Status | Fixed |
| Category | Cryptography |
| Component | camlsnark_c |
| Location | o1-labs/snarky/src/camlsnark_c/libsnark-caml/libsnark/misc/pedersen.cpp:133 |
| Impact | On certain architectures, attackers able to gather some knowledge of a target system will be able to predict Schnorr secret key values. |
| Description | Snarky generates a Schnorr secret key in the OCaml layer as follows: |

```
scalar schnorr_secret_key() {
  return scalar_field::random_element().as_bigint();
}
```

The `random_element` function that is called here comes from `libff`, the underlying C library used to perform some curve operations within Snarky. The `pedersen.cpp` file references the `mnt4` and `mnt6` implementations found within `libff`, and both of these implement the `random_element` roughly as below:

```
mnt4_G1 mnt4_G1::random_element()
{
    return (scalar_field::random_element().as_bigint()) * G1_one;
}
```

The underlying `scalar_field::random_element` function has multiple implementation paths depending on the type of the `scalar_field` that calls it; however, all of these differing field implementations use the same base class implementation to get a random field element. This is implemented as follows:

```
template<mp_size_t n, const bigint<n>& modulus>
Fp_model<n, modulus> Fp_model<n,modulus>::random_element() /// returns random ele
➔  ment of Fp_model
{
    /* note that as Montgomery representation is a bijection then
       selecting a random element of {xR} is the same as selecting a
       random element of {x} */
    Fp_model<n, modulus> r;
    do
    {
        r.mont_repr.randomize();

        /* clear all bits higher than MSB of modulus */
        size_t bitno = GMP_NUMB_BITS * n - 1;
        while (modulus.test_bit(bitno) == false)
```

```
        {
            const std::size_t part = bitno/GMP_NUMB_BITS;
            const std::size_t bit = bitno - (GMP_NUMB_BITS*part);

            r.mont_repr.data[part] &= ~(1ul<<bit);

            bitno--;
        }
    }
   /* if r.data is still >= modulus -- repeat (rejection sampling) */
    while (mpn_cmp(r.mont_repr.data, modulus.data, n) >= 0);

    return r;
}
```

This function generates a random point in the Montgomery representation, then presents it to the caller without much alteration. NCC Group confirmed that none of the other callers include any randomness, so any entropy in the final response is derived from the call to the `mont_repr.randomize()` function. Since `mont_repr` is a type alias for an appropriately-sized `bigint`, this function is a call to the `randomize` function in `bigint.tcc`. This function is as follows:

```
template<mp_size_t n>
bigint<n>& bigint<n>::randomize()
{
    static_assert(GMP_NUMB_BITS == sizeof(mp_limb_t) * 8, "Wrong GMP_NUMB_BITS va
➜   lue");
        std::random_device rd;
        constexpr size_t num_random_words = sizeof(mp_limb_t) * n / sizeof(std::r
➜   andom_device::result_type);
        auto random_words = reinterpret_cast<std::random_device::result_type*>(th
➜   is->data);
        for (size_t i = 0; i < num_random_words; ++i)
        {
                random_words[i] = rd();
        }

    return (*this);
}
```

This function pulls random data from the C++ `std::random_device` engine, which is not guaranteed to return cryptographically secure content. Old implementations of C++ libraries will return deterministic randomness from low-quality sources such as a Mersenne twister, which can be trivially broken.[1] In these cases, adversaries will be able to break the Schnorr secret generation process. Even the highest-quality and most modern implementations of the language include similar insecure sources as fallback generators.[2] As a result, these low-quality fallback implementations will be occasionally offered in unusual architectural configurations or even in typical ones due to the heterogeneous, unpredictable environment of

[1] For more information on how this particular randomness generator can be broken, see this video https://www.youtube.com/watch?v=f841Y7d3oDo.

[2] See line 159 of the LLVM implmentation at https://llvm.org/viewvc/llvm-project/libcxx/trunk/src/random.cpp?view=markup or line 267 of the GNU implementation at https://gcc.gnu.org/viewcvs/gcc/trunk/libstdc%2B%2B-v3/src/c%2B%2B11/random.cc?view=markup#l276.

computers, and in these cases Schnorr secret generation will be broken.

The `scalar_field::random_element` function sees a good deal of use in other parts of `libsnark-caml`. NCC Group did not investigate these, as they were not in the scope of this review. Nevertheless, any other secrets generated in this manner will have the same issues.

**Recommendation**

One option for the Snarky project is patching their version of `libff` to ensure it uses a robust source of secure randomness. Fixing this purely through passing a flag preferring one of the secure random number sources will not necessarily work, as architectures that do not support the relevant option may still pass through to insecure options. Instead, consider sourcing randomness from a secure construct like the libsodium `randombytes` function,[3] or otherwise by ensuring that the code uses and generates secure randomness in a portable way. An exemplar implementation of portable secure randomness can be found in the libsodium project's own implementation, available at https://github.com/jedisct1/libsodium/blob/master/src/libsodium/randombytes/sysrandom/randombytes_sysrandom.c.

Alternatively, it may be easier to change the `scalar_field::random_element` call to a different function. This can be via one of the paths suggested above, or, on modern Linux systems only, through the `getrandom` syscall with the `GRND_NONBLOCK` bit not set. This can be made more portable with equivalent `#ifdef`-guarded calls to `/dev/urandom/` on older Linux Machines and Apple devices, `getentropy` on BSD devices, and `CryptGenRandom` on WIndows devices.

**Client Response**

O(1) Labs has migrated this randomness generation subroutine to that of libsodium in the fix for issue 466 (commits `5ad0b9c`, `bf468ac`, and `721d70b`) as per NCC Group's suggestion. As a result, this issue has been fixed.

[3] Project hosted at https://github.com/jedisct1/libsodium

| | |
|---|---|
| **Finding** | **Potential Mishandling of Point Addition Edge Cases** |
| Risk | **Low** Impact: Low, Exploitability: Undetermined |
| Identifier | NCC-O1LB001-001 |
| Status | Fixed |
| Category | Cryptography |
| Component | snarky |
| Location | o1-labs/snarky/snarky_curve/snarky_curve.ml:360 |
| Impact | In a scenario where a prover acts on malicious input, the prover might be forced to generate an invalid proof that verifiers will reject, but the prover may not notice it. |

**Description**

The `snarky_curve` elliptic curve implementation uses a standard "short Weierstraß" curve. The formulas for adding two points $P_1 = (x_1, y_1)$ and $P_2 = (x_2, y_2)$ involve computing the slope $\lambda = (y_2 - y_1)/(x_2 - x_1)$; this operation does not work in the edge cases where $x_1 = x_2$, i.e. when $P_1 = P_2$ or $P_1 = -P_2$. The R1CS constraints generated from the addition operation duly verify that none of such additions is an edge case.

When doing a point multiplication operation (computing $kP$ for a given curve point $P$ and an integer $k$), the implementation uses a skew and a 2-bit window:

- An "unrelated base" $S$ point is used.
- The window contains $S$, $S + P$, $S + 2P$, and $S + 3P$.
- If $k$ is an $m$-bit integer (less than $2^m$, with $m$ even), then the multiplication operation really computes $kP + (2^m - 1)S$, and the point $(2^m - 1)S$ is subtracted at the end.

In general, $k$ is the output of a hash function, and may be larger than the curve order. Depending on the exact points $P$ and $S$, one of the involved additions may be an edge case. The current implementation, during proof generation, will not notice such an occurrence, and produce an invalid proof; verifiers will reject it, since the constraints will not be met.

As long as $k$ and $P$ are obtained "normally", this happens only with negligible probability. However, this raises the question of whether a specific context where the prover computes the point multiplication on externally provided values could be abused by a malicious attacker to force the prover to hit an edge case and generate an invalid proof. Whether such a context applies depends on how the Snarky library is used; it is a property of the overall protocol.

It can be shown that forcing an edge case requires knowledge of the discrete logarithm of $S$ relatively to $P$, i.e. the integer $z$ such that $S = zP$. Therefore, this potential attack is prevented with minimal effort if $S$ is obtained as the output of a random oracle over the point $P$. This feature is apparently planned but not implemented yet:

```
module Scaling_precomputation = struct
  type t = {base: Constant.t; shifts: Constant.t array; table: Window_table.t}

  (* TODO: Compute unrelated_base from g as
 unrelated_base = group_valued_random_oracle(gx, gy) *)
  let create ~unrelated_base base =
    let shifts = pow2s unrelated_base in
    {base; shifts; table= Window_table.create ~shifts base}
end
```

**Recommendation**

Implement the generation of $S$ from $P$ as a random oracle. The goal is to make the discrete logarithm of $S$ relatively to $P$ unknown to attackers; a simple method to do so is the following:

1. Use an encoding of $P$ as seed into a cryptographically secure PRNG (e.g. a XOF such as SHAKE).
2. Generate a sequence of $n$ bits (where $n$ is the bit length of the field) from the PRNG, and interpret it as an integer $x_s$.
3. If $x_s \geq q$, where $q$ is the field order, then loop to step 2.
4. If $x_s^3 + ax_s + b$ is not a quadratic residue in the field (with $a$ and $b$ being the curve equation parameters), then loop to step 2.
5. Compute $y_s$ as a square root of $x_s^3 + ax_s + b$.
6. Obtain one more bit from the PRNG; if that bit is 1, replace $y_s$ with $-y_s$.

This process yields a point $S = (x_s, y_s)$ whose discrete logarithm relatively to $P$ is fully unknown (insofar as the PRNG is indeed cryptographically secure).

Note: the above assumes that the curve has prime order. If the curve order is equal to $hr$ for a prime $r$ and another integer $h$ (known as the *cofactor*), then multiply the obtained point $S$ by $h$ to ensure that $S$ is part of the subgroup of order $r$.

**Client Response**

The client has remediated this issue as per NCC Group's suggestion at commit 2a85945. As a result, this is marked as fixed.

| | |
|---|---|
| **Finding** | **Out-of-Bounds Memory Write** |
| Risk | **Low** Impact: Low, Exploitability: Low |
| Identifier | NCC-O1LB001-006 |
| Status | Fixed |
| Category | Other |
| Component | Ledger App |
| Location | coda/ledger-coda-app/src/crypto.c:443 |
| Impact | Key generation results in an out-of-bounds memory write, resulting in potentially unexpected behavior and leakage of 16 bytes of the private key. |

**Description**

O(1) Labs has constructed a Ledger app that implements the transaction authorization scheme used in the Coda Protocol, based on Schnorr signatures, in low-level code for supported Ledger hardware wallet devices. NCC Group reviewed the implementation for its adherence to the signature scheme specified in the Coda documentation. In addition, NCC Group reviewed the implementation for any issues that could result in the exposure or loss of private key material, for example the secure generation of the nonce $k$ used in Schnorr signatures.

The Ledger app defines a function `generate_keypair` for deriving key material from the device's root seed. This function uses the built-in `os_perso_derive_node_bip32` function to derive a private key using the device hardware. Since Coda Protocol's keys are 48 bytes, but the derived secret is 32 bytes, subsequent calls to `os_memcpy` are used to combine the derived key with the "chain code":

```
// ...snip...
  os_perso_derive_node_bip32(CX_CURVE_256K1, bip32_path,
                             sizeof(bip32_path) / sizeof(bip32_path[0]),
                             priv_key, chain);
  os_memcpy(priv_key + 32, chain, 32);
  os_memcpy(priv_key + 64, chain, 32);
// ...snip...
```

`priv_key` is a `scalar`, which has a 48-byte capacity:

```
// ...snip...
#define scalar_bytes  48
// ...snip...
typedef unsigned char scalar[scalar_bytes];
// ...snip...
```

Thus, the call to `os_memcpy` will overwrite the memory boundary of the private key. 64 bytes are written, starting at `priv_key + 32`, resulting in the 48 bytes adjacent to `priv_key` being overwritten with the contents of `chain`. The impact of this finding is reduced, since the "chain code" is actually the output of HMAC with an unknown key—this is to say the chain code is pseudo-random, and therefore an attacker will be severely constrained in their ability to craft any effective memory corruption exploit to usurp the Ledger app's control flow. However, overwriting out-of-bounds memory may result in unexpected behavior. In addition, since the first 16 bytes of `chain` are included in the private key, writing `chain` to an out-of-bounds

location may result in partial leakage of the private key.

**Recommendation** Write only 16 bytes of `chain` to `priv_key + 32`.

**Retest Results** In commit 828660e, the write was changed to write only 16 bytes of chaincode, successfully addressing this finding.

| | |
|---|---|
| **Finding** | **Poseidon S-Box is Not a Bijection Under MNT Curves' Field** |
| Risk | **Informational** Impact: None, Exploitability: None |
| Identifier | NCC-O1LB001-002 |
| Status | Fixed |
| Category | Cryptography |
| Component | snarky |
| Location | o1-labs/snarky/sponge/sponge.ml:127 |
| Impact | Poseidon may not have the expected security properties when used with MNT curves. |

**Description**

A common class of zero-knowledge proof (ZKP) used in Coda Protocol proves that a user knows the preimage of a hash; for instance, a user might want to prove they know the preimage of Merkle tree nodes to show they are the originator of a transaction included in it. Using a cryptographic hash function such as SHA-256 for this purpose seems like a natural choice, but embedding information about these proofs into Coda Protocol's ZKPs would be extremely expensive. In R1CS, the circuit description format used by Coda Protocol, the number of multiplications in a particular program is generally predictive of the computational complexity of proof generation. This clashes directly against the design of nearly all modern cryptographic hashes, which tend to take advantage of the efficiency of bit rotation on common hardware; bit rotation is mathematically modeled as multiplication or division by two, meaning that computing the hash function is extremely complex. To sidestep this problem, the O(1) Labs team chose to use Poseidon,[4] a hash algorithm specifically designed to be both secure and efficient when expressed as an R1CS circuit.

One of the key choices for Poseidon's efficiency is its substitution box (S-box), which provides non-linearity (a key cryptographic security property) to the function. Poseidon's S-box is designed to be bijective, which is desirable as it

> ...ensures that all possible $2^n$ n-bit input vectors will map to distinct output vectors (i.e., that the s-box is a permutation of the integers from 0 to $2^n - 1$); this is a necessary condition for invertibility of the s-box (which may or may not be important, depending on the structure of the surrounding network) and ensures that all output vectors appear once (which guarantees that all possible input vectors are available to the next stage in the network).[5]

The S-box O(1) Labs has chosen will not be bijective if used with the MNT curve because it is not bijective over the prime field in which the MNT curves operate.

Snarky defines the core of the Poseidon algorithm as follows:

```
let block_cipher {Params.round_constants; mds} state =
  let sbox = to_the_alpha in
  let state = ref state in
  add_block ~state:!state round_constants.(0) ;
...
```

The S-box here is defined as the `to_the_alpha` function, which is itself defined as:

[4] A full description of this function can be found at https://eprint.iacr.org/2019/458.pdf.
[5] Taken from "The Structured Design of Cryptographically Good S-Boxes", available at https://link.springer.com/content/pdf/10.1007/BF00203967.pdf

```
let to_the_alpha x =
  let open Field in
  let res = square x in
  res *= res ; (* x^4 *)
              res *= x ; res
```

As can be seen here, Snarky makes use of the `HADESFIFTH` S-box, which is computed as $x^5$. The Poseidon paper notes that when using this S-box, it is only a bijection over the prime field $GF(p)$ in the case that $p \not\equiv 1 \mod 5$. However, the underlying field for MNT4 is defined by a prime that is in fact equivalent to one modulo five[6] .

Since $p \equiv 1 \mod 5$ there are 5 distinct $5^{th}$ roots of 1 modulo p, equal to $1, v, v^2, v^3$ and $v^4$ for some $v$. Then, only $\frac{(p-1)}{5}$ non-zero elements of $Z_p$ have a 5th root, and each that has a 5th root has 5 of them. In this case, the function is not a bijection. Since the proof of security relies on an S-box structure that is explicitly intended to be bijective, it does not necessarily apply any longer in this case.

The O(1) Labs team has informed NCC Group that the field over which the Poseidon S-box is calculated will change soon, so this issue has been graded as Informational to reflect that the system will likely not be at risk due to this.

**Recommendation** There are two avenues for fixing this issue. First, the S-box may be switched from `HADESFIFTH` to `HADESINVERSE`, which is calculated as $x^{-1}$. This will immediately solve the problem, but could result in a different performance profile. The other option would be to compute the Poseidon S-box over a different field. The included BN-128 and BN-382 curves appear to be suitable, with their primes both being equivalent to 3 modulo 5, making this a relatively pain-free switch if necessary.

**Client Response** The BN-based cycle now uses $x^{17}$ as the S-box. The O(1) Labs team reports that this is a more efficient S-box than those suggested above, and believes it should be equivalently secure in practice. This change can be seen at commit 2a85945.

[6] The prime, 475922286169261325753349249653048451545124879242694725395555128576210262817955800483758081, is equivalent to one modulo five, which can be observed from its decimal representation ending in a trailing one.

| | |
|---|---|
| **Finding** | **Future Developments May Impact Subgroup Security of Discrete-Log-Based Elliptic Curve** |
| Risk | **Informational** Impact: None, Exploitability: None |
| Identifier | NCC-O1LB001-007 |
| Status | Reported |
| Category | Cryptography |
| Component | Zexe |
| Location | N/A |
| Impact | Future protocol developments could lead to a case in which an attacker with the ability to perform fine timing measurements on a victim could result in a key leak. |
| Description | A small subgroup attack can be mounted on a discrete-logarithm-based cryptographic scheme that uses a prime-order group that is contained in a larger group of order divisible by small prime factors.<br><br>In particular, by forcing a protocol participant to carry out an exponentiation of a non-prime-order group element with a secret exponent, an attacker could obtain information about that secret exponent. This is possible if the protocol implementation does not check that the group element being exponentiated belongs to the correct subgroup and thus has large prime order.<br><br>Generally speaking, any addition to the protocol that would allow a malicious user to require a victim to exponentiate a given point over the group $G_2$ (such as a blind signature primitive) could result in a timing attack. This is a result of $G_2$'s possession of a cofactor that has a small subgroup of size 5, in which the discrete logarithm problem is easy. The attacker then proceeds as typical in a small subgroup attack, by repeatedly forcing the victim to exponentiate chosen points until the attacker can successfully deduce the underlying key (or a portion thereof). This attack is possible due to the underlying library used for exponentiation not checking for group membership prior to performing the operation.<br><br>It should be noted that this attack does not apply to the current iteration of the Coda Protocol (hence the Informational rating of this finding), but is noted for posterity. |
| Recommendation | If any future protocol additions require exponentiation of elements of $G_2$, ensure that the exponentiation function fails if the received point is not in the correct subgroup. |

| | |
|---|---|
| **Finding** | **Ledger Signing Does Not Abort When `k = 0`** |
| Risk | **Informational** Impact: High, Exploitability: None |
| Identifier | NCC-O1LB001-009 |
| Status | Reported |
| Category | Cryptography |
| Component | Ledger App |
| Location | `ledger-coda-app/src/crypto.c` L489 (sign) |
| Impact | Signing diverges from the specification, introducing the possibility of a fault injection attack. |
| Description | Coda Protocol uses an implementation of Schnorr-based signatures to authorize transactions. The implementation of signing is *deterministic*; the signature values are dependent solely on the message, public key, and private key. The scheme takes the following form:<br>• Compute $k = H(m + pk.x + pk.y + sk)$<br>• Compute $r = g^k$<br>• Compute $e = H(x + pk.x + pk.y + r.x + m)$<br>• Compute $s = k + e * sk$<br>Where $g$ is the generator of the prime-order subgroup, $sk$ is the private key, $pk$ is the public key, and the signature is the pair $(e, s)$.<br>The Coda Protocol documentation explicitly calls for a check that $k \neq 0$; however, this check is missing from the implementation. This represents a divergence from the specification.<br>Since the hash function $H$ is assumed to be a random oracle that chooses uniformly from $\mathbb{Z}_q$, the probability of $k = 0$ is $2^{-382}$. It is also assumed that $H$ is preimage resistant, therefore an attacker should not be able to choose a message $m$ for which $k = 0$. However, if a signature is produced with $k = 0$, this will immediately leak the private key. |
| Recommendation | Add the check called for in the documentation to the Ledger implementation. Alternatively, update the code with a comment to explain why it is not necessary. |

# Appendix A: Finding Field Definitions

The following sections describe the risk rating and category assigned to issues NCC Group identified.

## Risk Scale

NCC Group uses a composite risk score that takes into account the severity of the risk, application's exposure and user population, technical difficulty of exploitation, and other factors. The risk rating is NCC Group's recommended prioritization for addressing findings. Every organization has a different risk sensitivity, so to some extent these recommendations are more relative than absolute guidelines.

### Overall Risk

Overall risk reflects NCC Group's estimation of the risk that a finding poses to the target system or systems. It takes into account the impact of the finding, the difficulty of exploitation, and any other relevant factors.

**Critical** Implies an immediate, easily accessible threat of total compromise.

**High** Implies an immediate threat of system compromise, or an easily accessible threat of large-scale breach.

**Medium** A difficult to exploit threat of large-scale breach, or easy compromise of a small portion of the application.

**Low** Implies a relatively minor threat to the application.

**Informational** No immediate threat to the application. May provide suggestions for application improvement, functional issues with the application, or conditions that could later lead to an exploitable finding.

### Impact

Impact reflects the effects that successful exploitation has upon the target system or systems. It takes into account potential losses of confidentiality, integrity and availability, as well as potential reputational losses.

**High** Attackers can read or modify all data in a system, execute arbitrary code on the system, or escalate their privileges to superuser level.

**Medium** Attackers can read or modify some unauthorized data on a system, deny access to that system, or gain significant internal technical information.

**Low** Attackers can gain small amounts of unauthorized information or slightly degrade system performance. May have a negative public perception of security.

### Exploitability

Exploitability reflects the ease with which attackers may exploit a finding. It takes into account the level of access required, availability of exploitation information, requirements relating to social engineering, race conditions, brute forcing, etc, and other impediments to exploitation.

**High** Attackers can unilaterally exploit the finding without special permissions or significant roadblocks.

**Medium** Attackers would need to leverage a third party, gain non-public information, exploit a race condition, already have privileged access, or otherwise overcome moderate hurdles in order to exploit the finding.

**Low** Exploitation requires implausible social engineering, a difficult race condition, guessing difficult-to-guess data, or is otherwise unlikely.

## Category

NCC Group categorizes findings based on the security area to which those findings belong. This can help organizations identify gaps in secure development, deployment, patching, etc.

| | |
|---|---|
| **Access Controls** | Related to authorization of users, and assessment of rights. |
| **Auditing and Logging** | Related to auditing of actions, or logging of problems. |
| **Authentication** | Related to the identification of users. |
| **Configuration** | Related to security configurations of servers, devices, or software. |
| **Cryptography** | Related to mathematical protections for data. |
| **Data Exposure** | Related to unintended exposure of sensitive information. |
| **Data Validation** | Related to improper reliance on the structure or values of data. |
| **Denial of Service** | Related to causing system failure. |
| **Error Reporting** | Related to the reporting of error conditions in a secure fashion. |
| **Patching** | Related to keeping software up to date. |
| **Session Management** | Related to the identification of authenticated users. |
| **Timing** | Related to race conditions, locking, or order of operations. |

# Appendix B: Testing Methodology

NCC Group's overall strategy in this assessment relied largely on manual review of the program paradigm while looking for issues that the team had seen in implementations of constructs similar to those explored in this engagement. For each of these, the team looked for potential typos and logic issues that may occur in the source programming language, such as integer overflows, type confusion, or incorrect binding of `if` statements. Past these common checks, each construct has its own particularities requiring deeper study to ensure its security. These are detailed per-examined-primitive below:

## Compilation of Snarky to Rank-One Constraint Satisfiability (R1CS) System

At its core, the Coda Protocol leverages a system of zero-knowledge succinct non-interactive arguments of knowledge (zk-SNARKs, or just SNARKs), a type of cryptographic proof proving that a particular person has some special knowledge (for instance, that they control a specially-structured secret key) without revealing anything else. These are expressed by mathematical programs called circuits that can be expressed in a number of ways. The Coda Protocol has chosen to express these circuits as R1CS systems.

Rigorously defined, an R1CS system is an instance of the following problem[7]: given a vector $v \in F^k$ and three matrices $A, B, C \in F^{m \times n}$, can one augment $v$ to $z \in F^n$ such that $Az \cdot Bz = Cz$? In other words, the system is compiled of a set of requirements (constraints), all of which must have rank one (i.e. the constraint equation is expressed in linear combinations of variables). As an example, an R1CS argument that a variable $x$ is Boolean might be expressed as $x == 0 \| x == 1$.

In Snarky, O(1) Labs's approach is to map high-level OCaml types to R1CS constraints. In this way, O(1) Labs leverages the strength of OCaml's type system alongside the underlying R1CS variables to ensure that OCaml Snarky programs will automatically coerce themselves to R1CS arguments that are valid by virtue of their typing.

As a result, the validity of the translation from OCaml type constraints to R1CS constraints is a key concern, as this translation layer is responsible for the coherence and effectiveness of the programs made at higher levels. NCC Group tested these translations both statically and dynamically. The team statically tested the OCaml layer using the Mascot[8] static analysis tool, and also via manual review of the code for general soundness (i.e. ensuring that the R1CS constraint fully captured the same concept as its higher-level type constraint). The team also tested dynamically by attempting to create programs with invalid types. NCC Group tested the R1CS layer statically by ensuring that the emitted logic matched the design and examining the relevant code.

## Schnorr Signatures

As mentioned above, the Coda Protocol's consensus mechanism relies on SNARKs expressed as R1CS systems to police the validity of transactions and blocks. However, this means that all of the cryptographic signatures in the consensus protocol (e.g. signatures on the transactions themselves) must be validated by embedding it as a SNARK, whereas typical consensus schemes can validate their signatures directly. This presents an issue, as typical signature schemes like ECDSA have mathematical structure that does not mesh with that of the Coda Protocol's SNARKs. As a result, most common cryptographic signature schemes could not be computed and verified efficiently in the context of the Coda Protocol. As a result, O(1) Labs uses Schnorr signatures, which can be securely instantiated over any group with prime order in which the discrete logarithm problem is hard. By instantiating Schnorr signatures in the group used by the Coda Protocol's SNARKs, computations around the mathematical validity of the signatures come "for free", so the team only needed to add in constraints ensuring the validity of the signature structure itself.

To test Schnorr signatures' integration into Snarky, the NCC Group team needed to ensure that the implementation was correct, and that the scheme was secure over Coda Protocol's chosen parameters. The team was able to verify the correctness of the scheme through inspection of the code due to its simplicity, and the security of the scheme over the MNT4 and MNT6 parameters was supported by the literature. The more challenging portion was ensuring that the integration of the signature into the SNARK itself was secure; for this NCC Group needed to ensure the following:

[7] Definition taken from https://eprint.iacr.org/2018/828.pdf, section 1.2
[8] Hosted at http://mascot.x9c.fr/ (note this site is served over plain HTTP and should not be viewed over an unsecured Internet connection)

- the use of the weak formation of the Fiat-Shamir transform is valid for all cases[9]
- the verifier can ensure liveness of the signature
- the signature does not have malleability issues allowing re-submission of previous signatures

The weak Fiat-Shamir transform is valid because of users' publication of their public keys prior to producing Schnorr signatures claiming knowledge of their discrete logarithms (in the terms of the paper cited below for the discussion of weak vs. strong Fiat-Shamir transforms, the $X$ value is prepublished and cannot be selected by a malicious prover such that they can create a trivial proof). The signature's liveness is assured by the Coda Protocol's inclusion of a per-account incrementing nonce in the transaction body (which is what the Schnorr signature will be signing). Signature malleability can arise either from mathematical properties of the signature itself (e.g. adding the curve order to the $s$ value of an ECDSA signature as seen in early Bitcoin transactions[10]) or from properties of the encoding used to communicate the signature (e.g. the OpenSSL DER encoding issue in early secp256k1 signatures for Bitcoin[11])—fortunately, Schnorr signatures are provably non-malleable from a mathematical perspective[12]; while the encoding scheme itself is malleable, it cannot be abused to re-submit signatures due to the embedded per-transaction nonces.

## Verifiable Random Function (VRF)

The Coda Protocol's proof-of-stake consensus system uses a random stake-weighted selection of "slot leaders" (entities that execute some necessary computations for a particular round of consensus and produce parameters to be used in blocks generated in their slot). The Coda Protocol produces the provable pseudo-randomness used to select slot leaders via a VRF, a cryptographic construction that takes in arbitrary data and a key and produces cryptographically secure pseudo-randomness along with a proof that this pseudo-randomness was generated securely and fairly. Note that the distinction between randomness and pseudo-randomness is not relevant to the security of the protocol and is largely semantic.

The Coda Protocol's VRF implementation follows the outlines of the ongoing IETF standardization of ECVRF[13] (due to implementation concerns about efficiently embedding the resulting program into an arithmetic circuit, some specific algorithm choices differ). The RFC describes a set of concerns that come into play for a VRF implementation that wants to ensure full uniqueness, pseudo-randomness, and full collision resistance[14]; namely, an attacker that supplies a maliciously-crafted public key can potentially cause predictable output from the VRF.

The team examined the constraints around public key interpretation to ensure that an attacker cannot supply a public key that is not on the curve, that is a member of a small subgroup, or that is a point at infinity. In addition, the team performed dynamic analysis on the protocol to ensure that these potential edge cases are handled correctly in the case that an attacker broadcasted a malicious public key over the network on an altered or improvised wallet.

Finally, the team examined the VRF's capability to supply the required pseudo-randomness in the case that an attacker maliciously generates their secret key, as generally speaking it is not possible for a VRF to ensure that its output is pseudo-random under selection of malicious secret keys. Specifically, the team considered the potential of a malicious secret key choice to result in pseudo-randomness that would allow an attacker or group thereof to repeatedly choose themselves as slot leaders, potentially permitting malicious forking. The team considered many attack scenarios, primarily focusing on the case in which a cabal of attackers collaborated to choose specially-structured keypairs in order to artificially increase the odds that members of the cabal could act as slot leaders.

[9] The "weak" Fiat-Shamir transform does not include the statement in its challenge (i.e. the statement is not hashed to create the NIZK's equivalent of a two-party zero knowledge protocol's challenge), which can have undesirable effects in certain protocols. See https://eprint.iacr.org/2016/771.pdf for more details.
[10] See https://github.com/bitcoin/bips/blob/master/bip-0062.mediawiki for a description and fix of this issue
[11] *ibid.*
[12] This arises directly from the proof that Schnorr signatures are multi-user strongly unforgeable under chosen-message attacks given in https://eprint.iacr.org/2016/191.pdf
[13] At the time of this writing, draft 5 was the most recent update available. This can be found at https://tools.ietf.org/html/draft-irtf-cfrg-vrf-05
[14] See sections 3.1, 3.2, and 3.3 of the RFC linked in the previous footnote for more information about these properties.

## Elliptic Curve Arithmetic

The Snarky package fields an elliptic curve expressed by a short Weierstraß equation working on points in the affine representation. Traditionally, this implementation can suffer from problems if an attacker maliciously supplies the point at infinity, points that are not on the intended curve in the first place, or additionally in cases where they are computing on points $P_1, P_2 \mid \{P_1 = P_2 \parallel P_1 = -P_2\}$ (as the standard addition algorithm does not account for these cases). The NCC Group team inspected the code to ensure that these cases are handled correctly.

Additionally, the NCC Group team examined the choice of elliptic curve parameters used for the blockchain SNARK to ensure that they would be suitable for the intended application and that they would not impact the security of the various functions computed over them.

## Ledger Hardware Wallet Implementation

The Coda Protocol will be integrated onto the Ledger platform, a hardware wallet intended to securely store a user's cryptocurrency keys. The Coda Protocol re-implemented core primitives of the protocol to allow the device to integrate with the larger Coda ecosystem. The NCC Group team examined this re-implementation both for fidelity to the original implementations as well as security in its integration with the Ledger device. After ensuring that the two implementations were a matched pair via inspection, the team concentrated on traditional application security issues with the implementation with a focus on special issues that can arise on the Ledger platform.

The hardware wallet cooperates with a host over USB to sign and broadcast transactions. The job of the host is to construct transaction data and send this transaction data to the device, where the user can manually confirm details of the transaction using the device's on-board display and buttons. Once confirmed, the device signs the transaction and sends it back to the host, who then broadcasts the transaction. NCC Group reviewed the implementation of Coda's Schnorr signatures for any issues that could result in the exposure of key material or the bypass of the transaction confirmation flow. The implementation's handling of untrusted host data was checked for any potential memory corruption vulnerabilities that could result in the exfiltration of private key material or the surreptitious approval of a transaction crafted by a malicious host. The method used by the implementation to generate the critical Schnorr nonce $k$ was investigated. The methods used for private key derivation were also probed. Additionally, the methods used to generate public addresses were examined, ensuring that a malicious host could not generate an address for which the private key would be unrecoverable without secret information held by the host.

## Blockchain SNARK

As discussed, the Coda Protocol uses SNARKs to ensure the validity of overall blockchain state transitions. This allows for less-data-intensive blockchain participation, as O(1) Labs has implemented special SNARK constructions. These SNARKs leverage the concept of incremental computability to ensure that a user can validate the blockchain's entire history at any point using data that increases linearly in relation only to the number of transactions since the preceding block. Coda includes both a pairing-based and discrete-log-based implementation based on the MARLIN preprocessing SNARK.[15] One is a pairing-based construction instantiated using the Kate polynomial commitment scheme,[16] and the other is a discrete-logarithm-based construction instantiated using Halo's polynomial commitment scheme.[17] Each of these SNARKs (designated by O(1) Labs as "Tick" and "Tock" respectively) verifies proofs generated by the other and generates its own proofs that the other is expected to verify. Additionally, Tick has the ability to merge two Tock proofs into a single Tick proof. This simple property is the key to the succinctness of the Coda Protocol.

Coda Protocol emits single proofs of full blocks by structuring them into a tree of proofs with a root that carries the data of all leaves. This occurs as follows:

1. the Tick SNARK proves the validity of each individual transaction (note that NCC Group did not review this mode of Tick)
2. the Tock SNARK validates these and emits its own "wrapping" proof of the same statement

[15] The paper for this scheme is available at https://eprint.iacr.org/2019/1047.pdf
[16] This scheme is detailed in https://www.iacr.org/archive/asiacrypt2010/6477178/6477178.pdf
[17] This scheme is detailed in https://eprint.iacr.org/2019/1021.pdf

3. Tick can then validate the intersection of two of these wrapping proofs and emit a proof that is effectively a merger of the two wrapped proofs (i.e. validating the merger proof is equivalent to validating each of the wrapper proofs individually)
4. Tock then validates this merged proof and emits a new wrapper
5. Tick validates two wrappers and emits a new merged proof
6. Repeat steps four and five until left with a single Tick proof; this is the root of the tree, and the validation of this proof is equivalent to simultaneously validating all leaf transactions

An illustration of this structure can be found below:

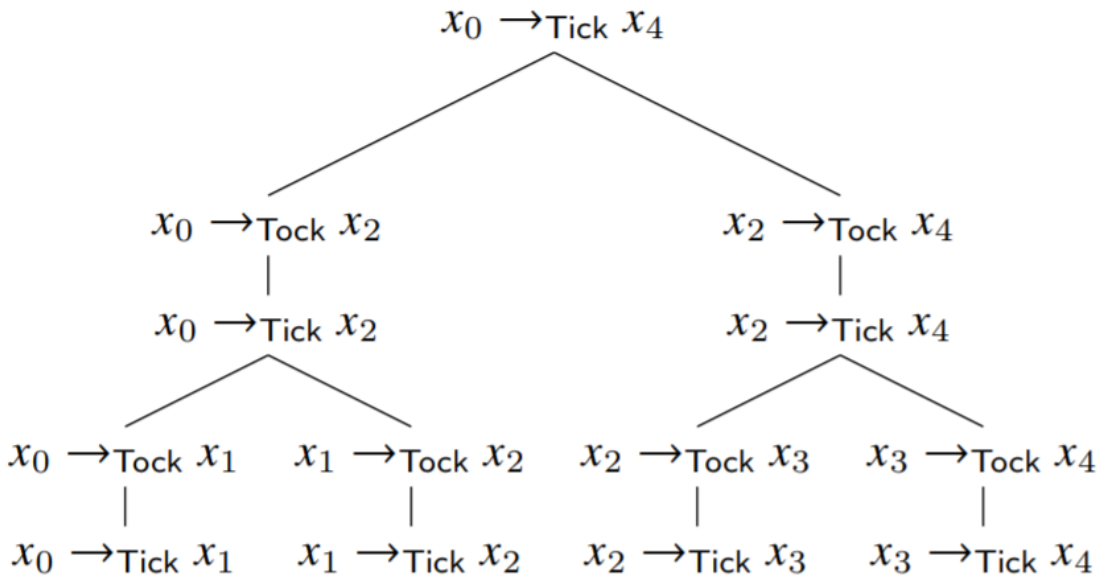


Figure 1: Reproduced from section 4.1.2 of "Coda: Decentralized Cryptocurrency at Scale"

The team evaluated the following assumptions of SNARK security:

- The implementations matched their theoretical counterparts
- No sensitive or bypass elements were included as part of the protocol
- Degenerate or edge case mathematical entities were handled appropriately

NCC Group reviewed the code side-by-side with the source papers as well as O(1) Labs's own work to ensure that there was a match between the implementation and the upstream research. Similarly, the team tracked all of the elements of the protocol that could potentially reveal unintended or excess information to the external world throughout the code base to ensure that they were never output. Finally, the team proceeded to ensure that all mathematical edge cases were successfully handled by the SNARKs. The team specifically checked for the following:

- All group members are verified to be in the appropriate subgroup
- Polynomials of inappropriate degree are not accepted
- Insertion of the group identity element is filtered or does not result in undesirable functionality
- Insertion of the field additive or multiplicative identity is filtered or does not result in undesirable functionality

NCC Group leveraged static analysis as much as possible, supplementing with a simulation of dynamic analysis for particularly complex pieces. For static analysis, the team examined the relevant portions of code to ensure that the appropriate security functionality existed and was faithfully implemented to the specifications of its source academic work. In some cases, NCC Group could not predict the interaction of the mathematical construct under study with

some of the edge cases discussed above. To increase overall confidence, the team created models of the relevant construct in Sage, then inserted appropriate elements as an approximation of dynamic analysis. However, the team could not be certain that the models were a one-to-one match to the source constructs.

NCC Group did not attempt to test or verify any mathematical claims in any source papers, including those produced by O(1) Labs. Instead, the team attempted to validate that the project was a faithful and secure implementation of those constructs in the assumption that the security properties asserted in these papers did in fact apply.

## Overall Protocol Review

The Coda Protocol's consensus protocol is largely implemented in two high-level functions: `update_var`, which advances the internal consensus state within the SNARK; and `select`, which implements the chain-selection rule. These are discussed separately below.

### Review of `update_var`

The Coda Protocol's consensus core relies on a single incrementally-computable SNARK $S$ that computes and verifies state transitions that update the blockchain state $\Sigma$, the set of pairs of ledger hash values and consensus states, and $T$, the set of blocks. Each block is a tuple consisting of a new (incrementally increasing) serial number, a record of its proposer's public key, the ledger state prior to the current block (represented as a state hash), a proof that this block is a valid state transition, a set of transactions, and the proposer's signature over all previous values. Each transaction is a tuple of the sender's public key, the receiver's public key, the amount to be sent, the sender's account nonce, and the sender's signature over all the preceding other than the sender's public key.

At a high level, the `update_var` function updates the current state of the blockchain via a transition that is specified by a new block proposed by the leader of the current slot. This transition updating $\Sigma$, implemented in `update_var`, is as follows:

1. Ensure that all transactions in the block are valid
2. Check that the block faithfully represents the current state of the ledger
3. Validate the current block's proof that it is a valid state transition
4. Verify the slot leader's signature over the information presented in this block
5. If all these work, update the ledger and consensus states, then return the new states

The team examined each of these sub-steps separately, and discusses the process behind each below.

#### Transaction Validation

The validation procedure for a Coda Protocol transaction succeeds if all of the following questions are answered in the affirmative:

1. Does the sender have sufficient funds in their account?
2. Does the sender's public key have the appropriate nonce?
3. Did the sender sign this transaction?

For the implementation of the first, the team was primarily checking to ensure that there were no tricks that could be pulled around the representation of currency amounts as integers. The Coda Protocol implements a custom signed integer type for tracking currency, which ensures that a person cannot perform actions such as sending "negative zero" (i.e. the two's complement of zero) and having this validate in transaction checking but resulting in the recipient being artificially credited. For the second issue, the team also ensured that there were no equivalent tricks surrounding nonces, but again these are tracked securely. NCC Group had already checked signature validation by this point (see the discussion of Schnorr signatures above).

#### Ledger State Validation

As discussed before, the ledger state is represented here as a state hash that is computed recursively over the set of tuples of prior ledger state hashes and their related consensus states. Fortunately, the Coda Protocol represents the current ledger state as a frozen hash and does not recompute it, which makes it challenging for an attacker to

manipulate. The NCC Group team checked that the equality relation did not allow for more than one value to be "equal" to itself,[18] but the strong typing of OCaml prevents these issues.

#### State Transition Validation

The Coda Protocol validates a state transition using the blockchain SNARK. In this paradigm, the prospective block is seen as a transition from one ledger/blockchain state to another whose validity is proven by a zero-knowledge proof embedded in the block. NCC Group's procedure in evaluating the mathematical portion of this construction is covered in the "Blockchain SNARK" section above.

#### Signature Validation

NCC Group's examination of signature validation can be found in the "Schnorr Signatures" section above.

### Review of `select`

As the final step for resolving forks in the protocol, the Coda Protocol implements chain selection as a `select` function. This function has two possible techniques to apply: a short-range fork rule and a long-range fork rule.

The function chooses the short-range rule when a fork is introduced that does not affect the block density distribution (i.e. an attacker has created their own blocks on top of prior blocks opportunistically); it simply chooses the longest chain, since such an attacker will not be able to backfill the number of blocks the legitimate chain has.

The function chooses the long-range rule at when the fork occurred long enough in the past that an attacker may have been able to affect block production. The long-range rule relies on a core Coda Protocol property: even after a successful fork, an attacker cannot alter consensus for a specific length of time[19] . Since a fork would presumably have fewer participants than the main chain, its block density is assumed to be lower during the period where a fork has occurred but has not affected consensus. Following from this observation, the long-range rule selects the chain with the highest minimum density in a specially chosen window following the fork. The testing methodology for both rules and the manner in which the blockchain chooses between them are discussed below:

#### Rule Selection

The selection of the rule to apply relies on the Coda Protocol's internal determination of time. Individual rounds of the consensus protocol (the base unit of blockchain time) are called slots, and these are in turn grouped into epochs (the first epoch length is determined at Genesis, and thereafter is dynamically determined). Intuitively, the rule selection algorithm is meant to apply the short-range rule to any forks whose size is less than or equal to two-thirds of an epoch[20] and to apply the long-range rule to all larger forks.

One issue arises from the intuitive description above: how can a blockchain node be aware of epoch timing if it tracks the ledger state with a simple hash? Requiring nodes to indefinitely maintain data about past epochs and slots would defeat the goal of producing a succinct blockchain in the first place. Coda Protocol solves this issue by storing two checkpoints of ledger state per epoch: a *start checkpoint* at the beginning of each epoch, and a *lock checkpoint* at the last block of the first two-thirds of the epoch (e.g. for an epoch of length 6, the start checkpoint would be taken from block 1 and the lock checkpoint would be taken from block 4).

Given these checkpoints, rule selection is as follows:

1. Call one candidate chain `cand_1` and call the other `cand_2`
2. If the lock checkpoint of `cand_1` and `cand_2` is the same, apply the short-range selection rule
3. Otherwise, apply the long-range selection rule

First, NCC Group manually checked the relevant piece of code to ensure that it correctly implemented the algorithm above. From this point, the team checked to ensure that the epoch checkpoints' equality function did not allow for

[18] Consider, for instance, that in JavaScript `[] == ![]`; such type confusion around equality would be catastrophic here.
[19] For more information on this, see section 7.3 of "Coda: Decentralized Cryptocurrency at Scale"
[20] Such a size is chosen because Coda Protocol's underlying consensus algorithm will not be appreciably affected by a malicious change within the first third of the epoch in which the fork occurs.

multiple values to be equal to one another.

#### Application of Short-Range Rule

As discussed above, the short-range rule returns the longer of the two chains. To test that this rule was applied correctly, NCC Group checked that the chain length was recomputed on checking (i.e. that a malicious fork could not simply report an arbitrary length) and that the type of the chain length safely implemented comparison (e.g. not allowing for representation such that a chain could claim a length equivalent to the two's complement of zero, then have this be interpreted as a large unsigned integer in comparison).

#### Application of Long-Range Rule

At a high level, the long-range rule attempts to maintain a space-efficient record of minimal block densities within a window structured such that the effect of a malicious fork's distortion is captured, even if the fork itself is not.

The first concept to tackle is the dynamic window structuring: the Coda Protocol employs a novel construct called a $\nu$-shifting $\omega$-window. In this construct, $\nu$ is called the shift parameter, and $\omega$ is known as the window-length parameter. The goal for this construct is to ensure that the window is slightly larger than the rate at which new non-empty blocks are produced (measured in slots), and that it is shifted to appropriately capture new slots when the old ones pass out of scope for consideration of forking. In practice, this is implemented via a set of sub-windows of length $\nu$; when the oldest passes out of scope, a new sub-window is created.

This structure is defined by the `shiftWindow` algorithm, which takes the existing set of sub-windows and a vector of new Coda Protocol consensus slots, updating them to fit the new slots. This occurs as follows:

1. First, set the minimum density to the lowest among the combination of new slots and slots existing in current sub-windows
2. Advance all sub-windows by the shift amount, allowing the oldest $\nu$ to pass out of scope
3. Return the new set of sub-windows

NCC Group first examined the code to ensure the function was implemented faithfully. Following this, the team brainstormed potential implementation issues and checked for them. Particularly, that the density of each slot is re-calculated prior to its insertion in the appropriate sub-window, to ensure that a malicious fork could not arbitrarily submit inaccurate densities and that the window density type did not allow for any type confusion in its underlying container that could affect the calculation of the minimum. Finally, the team ensured that there were no ways to alter $\nu$ such that it did not accurately reflect the protocol window.

The team moved on to examining the overall implementation of the chain rule after vetting the `shiftWindow` algorithm. This consisted solely of ensuring that the algorithm was faithfully executed, as the types and validity for the underlying information had already been verified in the previous step.