Zcash Overwinter Consensus and Sapling Cryptography Review

Zcash

January 30, 2019 – Version 1.3

Prepared for

Jack Grigg
Zooko Wilcox
Daira Hopwood
Nathan Wilcox
Benjamin Winston

Prepared by

Thomas Pornin
Aleks Kircanski
Mason Hemmel
David Wong
Janet Ghazizadeh
Mathias Hall-Andersen
Javed Samuel

Document Change Log

Version	Date	Change
1.0	2018-05-18	Initial report provided to client
1.1	2018-09-11	Updated report provided to client after retest
1.2	2019-01-24	Updated after public technical review
1.3	2019-01-31	Ready for public release

Executive Summary

In the spring of 2018, The Zerocoin Electric Coin Company engaged NCC Group to perform a two-pronged review of recent changes to the Zcash cryptocurrency. The first prong focused on updates to the Overwinter consensus code, such as architectural changes facilitating future network upgrades, and new features, such as transaction expiry. The second prong concentrated on the implementation of the cryptographic primitives used in the cryptocurrency's Sapling release. This release brings a number of changes to Zcash, including changes to core cryptographic components such as the underlying elliptic curves, shielded transaction structure, and signature scheme.

The Overwinter portion of the engagement consisted of 20 person-days split between two consultants, taking place between March 26 and April 6, 2018. The cryptographic review of the Sapling primitives' implementation consisted of 20 person-days split between four consultants. It ran from May 7 to May 18, 2018. The changes to the Overwinter portion are written in C++, whereas Sapling cryptographic primitives are implemented in Rust. Retesting was completed on September 4 and 5, 2018.

Scope

Overwinter Consensus Phase

The Overwinter review covered the entire code-base but focused on the following pull requests, chosen by the Zcash team as representative of the Overwinter changes:

- 2898
- 2925
- 2903
- 2463
- 2919
- 2874

The review included manual code review, searching for vulnerabilities specific to public block chain implementations as well as more general application security issues.

Sapling Cryptography Phase

As for the Sapling cryptographic review, the primary focus was on the following elements:

- The core Sapling repository, which implements primitives used in the Sapling release, such as variants of the Pedersen hash, Jubjub curve, Spend and Output circuits.
- Version 2018.0-beta-19 of the Zcash Protocol Specification document was used as primary documentation on the changes introduced by the Sapling release, when compared with the Sprout release.

Side channel attacks (e.g. timing attacks) were explicitly out of scope, since operations that manipulate secret values are normally performed on offline systems or in an asynchronous way, which is considered to make such attacks impractical.

Key Findings

- **Curve BLS12-381 Security Is Less Than 128 Bits.** The newly introduced curve BLS12-381 used inside zk-SNARKs does not offer a 128-bit security level. Recent public research[1] suggests that the curve achieves a security level between 117 and 120 bits at most. It should be noted that while this fact runs contrary to some stated security goals, it does not represent a practical threat. More details can be found in finding NCC-Zcash2018-004 on page 8.
- **RedDSA is not SURK-CMA.** The Zcash protocol specification requires that RedDSA/RedJubjub, a newly introduced signature scheme with re-randomizable keys, satisfies the *Strong Unforgeability with Re-randomized Keys Under Adaptive Chosen Message Attack* (SURK-CMA) property. However, finding NCC-Zcash2018-009 on page 10 shows the proposed signature scheme in fact does not satisfy this security property.
- **DoS Through Transient Chain View Differences.** An attacker could have a legitimate participant in the Zcash network banned by sending to the victim a flow of transactions that are on the brink of expiration. If the victim node forwards the transactions to other nodes and other nodes consider the transactions expired, the network will assess the victim node a non-zero DoS penalty, potentially leading to banning. This is shown in finding NCC-Zcash2018-001 on page 6.

[1] https://hal.archives-ouvertes.fr/hal-01534101/file/main.pdf

Dashboard

Target Metadata

Name	Zcash Overwinter Consensus and Sapling Cryptography Review
Type	Bitcoin-forked cryptocurrency
Platforms	C++, Rust
Environment	Zcash's "Sapling" Cryptography

Engagement Data

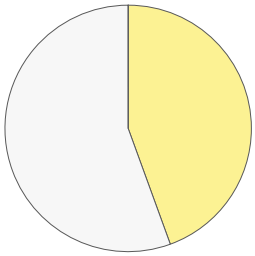
Type	Protocol and code review
Method	Code Assisted
Dates	March 26, 2018 to May 18, 2018
Consultants	6
Level of effort	40 person-days

Targets

Core Sapling Repository	https://github.com/zcash-hackworks/sapling-crypto/tree/5687acfaf83438a993fccc14ab487b67e4afbc68
Zcash Protocol Specification	https://github.com/zcash/zips/blob/7e0b51011a5fcf021e40a4c7882da54f1d6627fa/protocol/sapling.pdf

Finding Breakdown

Critical Risk issues	0
High Risk issues	0
Medium Risk issues	0
Low Risk issues	4
Informational issues	5
Total issues	**9**



Category Breakdown

Cryptography	3	
Denial of Service	1	
Other	4	
Uncategorized	1	

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 23.

Title	Status	ID	Risk
Asynchronous Chain Updates Can Lead To Network Bans	Fixed	001	Low
Curve BLS12-381 Security Is Less Than 128 Bits	Reported	004	Low
RedDSA / RedJubjub Are Not Strongly Unforgeable With Re-Randomized Keys	Fixed	009	Low
Bit Ordering Convention Leads to Mismatches and Confusion	Reported	010	Low
Mempool Not Properly Cleared If Connecting or Disconnecting Tips Fails	Reported	002	Informational
Private Key Decoding Has Limited Interoperability	Reported	003	Informational
Discrepancies Between Specification and Circuit Implementation	Reported	007	Informational
Typographical Inconsistencies in Zcash Specification	Reported	008	Informational
Bias in Random Field Element Generators for Underlying Representation	Reported	011	Informational

Finding Details

Finding	**Asynchronous Chain Updates Can Lead To Network Bans**
Risk	Low Impact: Low, Exploitability: Medium
Identifier	NCC-Zcash2018-001
Status	Fixed
Category	Denial of Service
Component	P2P Protocol
Location	`main.cpp`, lines 874, 894, 900.
Impact	Maliciously crafted transactions may induce victim nodes to reject each other and apply temporary banning (24 hours), disrupting communications.
Description	Zcash nodes exchange transactions and blocks in a peer-to-peer network. When a node obtains a transaction and deems it valid with regards to internal rules, it forwards it to its peers. However, if an *invalid* transaction is received by a node, then the node rejects it, and may apply a penalty on the sending node (through calls to `CValidationState::DoS()`). A total penalty counter is maintained for each known peer; when that counter reaches a given threshold, the faulty peer is forcibly disconnected, and banned from reconnecting for a configurable amount of time. By default, the penalty threshold is 100; penalties are 100 (major violations), 10 (minor violations), or 0. The default ban time is 24 hours. Most of the checks on transactions are intrinsic (e.g. proper encoding format). However, some checks are *contextual*: they depend on the current block chain height. For instance, Overwinter transactions may have an "expiry" field that indicates the chain height at which the transaction ceases to be a valid candidate for inclusion in new blocks. Nodes maintain their notion of the current block chain height based on the blocks they receive, and the consensus rules ensure that all nodes that follow the same rules will eventually converge on the same chain. However, there can be transient discrepancies between the views of the current chain by the nodes, if only because simultaneity of block reception is not guaranteed. As a new block propagates through the network, nodes update their view of the block chain, but there will routinely be a small time window, possibly of several seconds, during which two directly connected nodes will have distinct notions of the current chain height. During that window, a given transaction may be acceptable by one node but not by its peer, because of contextual checks that depend on the chain height. This allows a **denial-of-service attack** that works as follows: • The attacker sends many transactions to the victim node. The transactions are currently valid, but will become invalid when the next chain block is mined. • The victim node forwards the transactions to its peers. • If a peer node has already received the next block, then it will reject the transactions and apply a penalty on the victim node. The penalty for contextual rules is typically 10, and the threshold is 100; therefore, ten malicious transactions are sufficient to trigger a 24-hour ban. NCC Group found two types of situations where such an attack is possible: • **Transaction Expiry:** An Overwinter transaction can be tagged with an expiry height; when the block chain height reaches that value, the transaction is no longer acceptable for inclusion in blocks. An attacker can thus choose to send to the victim node a stream of transactions that are on the brink of expiration.

- **Protocol Upgrade:** Before the activation height of the new protocol (Overwinter), pre-upgrade transactions (v2) are deemed acceptable, but post-upgrade transactions (v3) are not. When the height is reached, the situation is reversed. The attacker can then send a stream of v2 transactions to a victim node, hoping for its peers to obtain the next block slightly earlier; alternatively, the attacker can send a stream of v3 transactions to a victim node that just obtained the new block, in case its peers are lagging.

The attack on transaction expiry can be performed on every new block; on the other hand, the protocol upgrade is a one-time opportunity for the attacker, and is thus unlikely to result in prolonged disruption. However, the latter might occur by chance, without any actual attack, if a sufficient number of v2 transactions are transferred at the wrong time.

It should be noted that a similar attack situation might have occurred with the "lock time" mechanism, except that a penalty of value zero is applied,[2] thereby not contributing to the banning threshold:

```
    if (!CheckFinalTx(tx, STANDARD_LOCKTIME_VERIFY_FLAGS))
        return state.DoS(0, false, REJECT_NONSTANDARD, "non-final");
```

The penalty/ban mechanism was added to the Bitcoin code base in 2011[3] as a generic mechanism aiming at reducing the impact of DoS attacks. At that time, it was already noted that duplicate transactions shall not trigger a non-zero penalty, to prevent an attacker from inducing bans by injecting conflicting transactions at different nodes in the peer-to-peer network. Currently, the only possible attack situations NCC Group found are related to Overwinter, and do not apply to the Bitcoin network.

Recommendation

The penalty applied for contextual violations should be set to zero, at least for off-by-one errors—when a transaction is rejected with regards to the current height *h*, but would have been accepted with height *h*+1 or *h*-1. Take care NCC Group does not advocate accepting more transactions, but only applying a lower penalty when an invalid transaction is rejected for contextual reasons.

Retest Results

Commit 473a1132419d05911933ac0095b207c4e2fc59f3 sets the penalty level to `0` when receiving a transaction that has just expired, and would have been acceptable if the height had been one less than its current value. This matches NCC Group recommendation, and addresses the issue with expired transactions. Zcash deemed the similar issue on protocol upgrade, which may happen only once at protocol switch time, too limited in its possible consequences to warrant any action.

[2] https://github.com/zcash/zcash/blob/b466c1c90cb7a3f01f40dbdde46dde36dc1820b8/src/main.cpp#L1205
[3] https://github.com/bitcoin/bitcoin/pull/517

Finding	**Curve BLS12-381 Security Is Less Than 128 Bits**
Risk	Low Impact: Medium, Exploitability: None
Identifier	NCC-Zcash2018-004
Status	Reported
Category	Cryptography
Component	Sapling
Location	New pairing-friendly curve.
Impact	The selected new curve (BLS12-381) does not achieve the targeted "128 bits" security level.

Description

The zk-SNARKS internally operate over a pairing-friendly curve. Such a curve is defined over a finite field of order q, and allows defining two groups G_1 and G_2 of prime order r; group G_1 is typically the subgroup of the points of r-torsion in the curve over field $\mathbb{F}_q$. The pairing is a bilinear function that maps an element of G_1 with an element of G_2 into an element of a third group G_3, which is the group of r-th roots of unity in the finite field extension $\mathbb{F}_{q^k}$ for a given integer k called the *embedding degree*. For the curve to be deemed secure, discrete logarithm (DL) must be hard in all three groups. In particular:

- Order r must have size at least $2n$ bits if the target security level is "n bits", since there are generic DL solving algorithms that work in cost $O(\sqrt{r})$ for any group of size r (e.g. Pollard's rho algorithm[4]).
- Finite field extension order q^k must be large enough to defeat index calculus methods, in particular based on the number field sieve.[5]

Zcash currently uses a Barreto-Naehrig curve such that q and r have sizes about 254 bits, for a DL solving cost of 2^{127} operations, and the embedding degree is $k = 12$. The general NFS algorithm would have cost slightly above 2^{128} in $\mathbb{F}_{q^{12}}$. Thus, when first defined, that curve was rated as achieving security level at least "127 bits".

That assertion relied on the idea that the general number field sieve (GNFS) was the most efficient algorithm for DL in $\mathbb{F}_{q^k}$. However, in late 2015, Kim and Barbulescu[6] found that the specific structure of q, inherent to the generation method used by Barreto and Naehrig, could be exploited into a faster DL solving algorithm called SexTNFS (special extended tower NFS). Menezes, Sarkar, and Singh[7] then used SexTNFS to estimate the actual security level of 256-bit BN curves at around 110 bits. This prompted Zcash to choose a new curve, in the "BLS12" family, that would restore the 128-bit security level.[8] The chosen curve has the following characteristics:

- Field order q is a 381-bit prime.
- Curve subgroup order r is a 255-bit prime.
- Embedding degree is still $k = 12$; the Menezes-Sarkar-Singh estimate of security of DL in the field extension is then about 131 bits.

However, in December 2017, Barbulescu and Duquesne published a new analysis[9] that stud-

[4] https://en.wikipedia.org/wiki/Pollard%27s_rho_algorithm_for_logarithms
[5] Initially invented by Pollard for factoring big integers, the number field sieve algorithm was adapted to solving discrete logarithms by Gordon in 1993.
[6] https://eprint.iacr.org/2015/1027
[7] https://eprint.iacr.org/2016/1102
[8] https://blog.z.cash/new-snark-curve/
[9] https://hal.archives-ouvertes.fr/hal-01534101/file/main.pdf

ies in more details the different phases of SexTNFS; in particular, they point out some sources of imprecision in the work of Menezes, Sarkar, and Singh. The revised attack costs are then lower:

- 256-bit BN curves now rate at about 100 bits of security instead of 110 bits.
- The new BLS12-381 curve would be estimated to achieve between 117 and 120 bits at most, falling short of the initially stated target level of 128 bits.

Barbulescu and Duquesne suggest new parameters for 128-bit and 192-bit security levels. In particular, for 128-bit security with a BLS12 curve, the base field should have order of at least 460 bits.

In practical terms, none of these curves are breakable with existing or foreseeable technology, not even the current 254-bit BN curve used by Zcash (even at "100-bit" resistance, it is still stronger than, for instance, RSA-2048). While not achieving "128-bit security", BLS-381 is still virtually one million times stronger than BN-254 (corresponding to the extra 20 bits of security), which ought to be robust enough for the use case of Zcash. The value of reaching "128-bit" is mostly psychological.

Recommendation

The new analysis by Barbulescu and Duquesne shows that BLS-381 cannot be truthfully advertised as offering "128-bit security". It supports, however, assertions that BLS-381 is substantially stronger than BN-254, which is already unbreakable with existing technology and algorithms.

Since SexTNFS is still an active research subject, and results are very recent, it is conceivable that even the revised estimates are not definitive. In order to better support potential improvements on SexTNFS, NCC Group recommends, as a long-term goal, studying a further switch to a new curve selected with a margin of safety. Performance considerations imply that the curve subgroup order should have size 256 bits or so; thus, a new long-term curve would need an embedding degree of more than 12. Candidates include the KSS curves[10] with embedding degrees 16 or 18, as well as BLS24 curves.

[10] https://eprint.iacr.org/2007/452

Finding	**RedDSA / RedJubjub Are Not Strongly Unforgeable With Re-Randomized Keys**
Risk	Low Impact: Undetermined, Exploitability: Undetermined
Identifier	NCC-Zcash2018-009
Status	Fixed
Category	Cryptography
Component	Sapling
Location	Zcash specification, sections 4.1.6.1 and 5.4.6.1.

Description

Section 4.1.6.1 defines the concept of a signature scheme with re-randomizable keys. For a given private key *sk*, a new private key can be derived by the adjunction of a randomization value α. Signature verification then works over the randomized public key, which can be derived from the core public key with the randomization value α. Re-randomizable keys are used to support spending authorization signatures while keeping spending statements unlinkable to actual signer identities. The security requirement for such a signature scheme is called *Strong Unforgeability with Re-randomized Keys under adaptive Chosen Message Attack* (SURK-CMA); it is described in the specification with the following game:

- A given signature oracle knows the private key *sk*.
- The attacker may submit arbitrary queries (m, α) to the oracle, where m is a message to sign, and α is a randomization value.
- To each query, the oracle responds with a signature σ computed over the message m, using the randomized private key obtained by combining *sk* and α.
- The oracle records pairs (m, σ) (the message from the query, and the obtained signature).
- The goal of the attacker is to obtain a triplet (m', α', σ') such that:
  - σ' is a valid signature on m' when verified against the public key randomized with α'.
  - The (m', σ') pair is not part of the list of queries recorded by the oracle.

The attacker wins if there was no query containing m' such that the oracle returned signature σ'. However, a query may have contained m' but yielding a signature different from σ'; or the oracle may have returned σ' but for a message different from m'. Note that randomization parameters α are freely chosen by the attacker, and are not recorded by the oracle.

Section 5.4.6 defines an incarnation of Schnorr signatures called RedDSA, which supports re-randomization. RedJubjub is RedDSA, when using the Jubjub curve as underlying group. Section 5.4.6.1 specifies that RedJubjub is to be used as signature scheme with re-randomized keys, in the sense of section 4.1.6.1.

However, **RedDSA is not SURK-CMA**. Here is a (simplified) description of RedDSA:

- The algorithm uses a group $\mathbb{G}$ that has a subgroup of prime order n. Point $\mathcal{P}$ is a conventional fixed generator for the prime order subgroup. Subgroup elements can be encoded into sequences of bits with a deterministic injective mapping. A hash function H works over arbitrary binary inputs, and produces a value modulo n as output.
- For private key *sk* and randomization value α:
  - The "raw" public key is: $Q = [sk]\mathcal{P}$
  - The re-randomized private key is: $sk_\alpha = sk + \alpha \bmod n$
  - The re-randomized public key is: $Q_\alpha = [sk + \alpha]\mathcal{P}$
- For message m, private key sk, and randomization value α, the signature σ is computed as follows:

  - An integer r is chosen randomly and uniformly in the $[1..n-1]$ range.
  - Let: $R = [r]\mathcal{P}$
  - Let: $s = (r + H(R \parallel m)(\mathsf{sk} + \alpha)) \bmod n$
  - Signature is: $\sigma = (R, s)$
- Signature verification uses message m and randomized public key Q_α (note that Q_α can be computed from Q and α without knowledge of the private key *sk*). The signature $\sigma = (R, s)$ is accepted if the following equation holds: $[s]\mathcal{P} = R + [H(R \parallel m)]Q_\alpha$

Suppose that a signature oracle is used, as in the definition of SURK-CMA. The attacker may then win the game in the following way:

- The attacker submits a query (m, α), and obtains the signature $\sigma = (R, s)$.
- The attacker chooses a non-zero value β modulo n, and computes the following values:
  - Let: $m' = m$
  - Let: $\alpha' = \alpha + \beta \bmod n$
  - Let: $\sigma' = (R, s')$ where $s' = s + H(R \parallel m)\beta \bmod n$

It can be verified that σ' is a valid signature for message m', relative to the public key re-randomized with α':

$$\begin{aligned} [s']\mathcal{P} &= [r + H(R \parallel m)(\mathsf{sk} + \alpha) + H(R \parallel m)\beta]\mathcal{P} \\ &= [r]\mathcal{P} + [H(R \parallel m)(\mathsf{sk} + \alpha + \beta)]\mathcal{P} \\ &= R + [H(R \parallel m)]Q_{\alpha'} \end{aligned}$$

However, with overwhelming probability, σ' is distinct from σ (because $s' \neq s$). Thus, the pair (m', σ') is not part of the pairs recorded by the oracle. Therefore, RedDSA (and thus RedJubjub) is not SURK-CMA.

Exact impact is undetermined. The attack above does not contradict *existential unforgeability*, in which the oracle records only messages m, and the attacker may win the game only if finding (m', α', σ') such that m' was never submitted to the oracle. However, the Zcash specification (section 4.1.6.1) insists on requiring strong unforgeability:

> Note that we require Strong Unforgeability with Re-randomized Keys, not Existential Unforgeability with Re-randomized Keys (the latter is called "Unforgeability under Re-randomized Keys" in [FKMSSS2016, Definition 8]).

The attack above produces a new signature value σ' for a message m that was signed by the oracle; the message itself is unmodified. Since re-randomized signatures are used for spending authorization, such an attack will not result in a forged transaction. Moreover, in Zcash, the signed data contains a copy of Q_α, which would prevent that exact alteration from being successfully applied in practice.

Retest Results

On September 5, 2018, a new version of the protocol specification (2018.0-beta-30) and the Zcash implementation (Git tag v2.0.0 and dependencies) was checked for a fix of this issue. The new protocol amends the definition of RedDSA to now compute $H(R \parallel vk \parallel m)$ (where *vk* is an unambiguous representation of the public key Q_α) instead of $H(R \parallel m)$. This addition prevents the attack described above, and thus addresses the issue.

It must be noted that while the old protocol did not mandate inclusion of the public key as part of the hash function input in RedDSA, its uses within the Zcash protocol (for *Spend*

Authorization Signatures and *Balance and Binding Signatures*) already added the encoding of the public key as a prefix to the signed data, which was functionally equivalent. The v2.0.0 version of Zcash internally follows the old convention, in that the RedDSA implementation expects the caller to have already prefixed the encoded public key; in that sense, the internal structure of the Zcash code does not match the formal description in the specification, but the computed values and externally observable behavior are in accord with it.

The RedDSA verification was also slightly altered, in that the final verification equation was transformed from:

$$[s]\mathcal{P} = R + [H(R \parallel vk \parallel m)]Q_\alpha$$

into:

$$[h](-[s]\mathcal{P} + R + [H(R \parallel vk \parallel m)]Q_\alpha) = \mathcal{O}$$

where h is the curve cofactor ($h = 8$ for Jubjub). This change was done in order to allow for optimized batch verification of several signatures, a process described in a new section B.1 in the protocol specification. The batch verification uses a linear combination of curve points with random coefficients; the multiplication by the cofactor is necessary to make the batch verification reliable (otherwise, maliciously crafted signatures may induce non-deterministic acceptance, which would break consensus rules). Non-batch verification must then also use the cofactor, so that batch and non-batch versions always agree.

Use of the cofactor is permitted in the specification of EdDSA,[11] from which RedDSA is inspired; it does not diminish security. **NCC Group recommends** that explicit notes be added in the specification (sections 5.4.6 and B.1) to make the use of the cofactor mandatory, in order to ensure consensus: Indeed, many existing implementations of EdDSA do not use the cofactor, because it does not matter in most situations where signatures are used, but Zcash requires all implementations to fully agree on which signatures are acceptable.

[11] Daniel J. Bernstein, Niels Duif, Tanja Lange, Peter Schwabe, Bo-Yin Yang, *High-speed high-security signatures*: https://ed25519.cr.yp.to/ed25519-20110926.pdf

Finding	**Bit Ordering Convention Leads to Mismatches and Confusion**
Risk	Low Impact: Low, Exploitability: None
Identifier	NCC-Zcash2018-010
Status	Reported
Category	Other
Component	Sapling
Location	src/group_hash.rs:36
Impact	The little-endian ordering of bits within octets is opposite to widespread conventions, which leads to slight inefficiencies, confusion, and mismatches between specification and implementation.

Description

The Zcash specification (section 5.2) defines encoding rules for elements used in Zcash, which fall in roughly three categories: bit strings, octet[12] strings, and big unsigned integers (which are usually modular integers). *Endianness* is the name for the ordering convention of sub-elements within a bigger element. In practice, endianness will be defined at two levels: bits within octets, and octets within big integers. The general rule in Zcash is to use little-endian ordering for both levels. While little-endian ordering of octets within big integers corresponds to the natural encoding rules of many modern architectures (in particular x86), the little-endian ordering of bits within octets is opposite to widespread conventions, which leads to slight inefficiencies, confusion, and mismatches between specification and implementation.

As an illustration, the MD5 hash function is normally defined as using little-endian encoding; this applies to interpreting sequences of octets into 32-bit integers. However, it uses big-endian encoding of bits within bytes: the MD5 padding rule for the last block involves adding a bit of value 1, followed by bits of value 0, to the input bit sequence; when expressed in octets, the first octet to append has value 0x80, which means that the first bit in the padding sequence is most significant within the first octet, not least significant. This rule is pervasive in most cryptographic standards and related areas; e.g. it also corresponds to the ordering of bits within a DER-encoded ASN.1 `BIT STRING` value.

The main consequence of the use of the little-endian convention at the bit level is that, when decoding incoming data as a big integer, or encoding it back, there must be a step by which bits are moved within each byte value (or within a larger word). In the `sapling-crypto` package, this is usually done with the `swap_bits_u64()` function, see `src/util.rs:5`. This is used, for instance, in `read_scalar()`, `src/redjubjub.rs:11`. The input sequence of bits is received as a sequence of octets (`u8` values), which is interpreted in full big-endian convention (both at bit and octet levels) and decoded into a sequence of 64-bit words; that sequence is then reversed, as well as the sequence of bits within each word.

Since this bit-swapping step is relatively inefficient and inconvenient to implement, the Zcash specification and implementation tend to avoid it when possible, either through exceptions in the specification, or through coalescing operations in the implementation. An example of an exception is in section 5.4.1.1: the SHA-256 function is defined by NIST[13] as operating on sequences of bits, which are decoded with full big-endian convention (at bit and octet levels); to avoid the bit-swapping operation, the Zcash specification states that the input bits are

[12] In the Zcash specification, the generic term "byte" is used for an octet; however, the conversion functions use "OS" to designate a string of bytes.
[13] FIPS 180-4, *Secure Hash Standard*, https://csrc.nist.gov/publications/detail/fips/180/4/final

assumed to have already been converted to an octet string (i.e. with big-endian convention at the bit level), and SHA-256 operates on octets.

Compounding the issue is the fact that encoding rules are not enforced by the Rust type system, and in fact traverse abstraction layers. For instance, the `pairing` library defines both a `PrimeField` trait, which defines a given field, and a `PrimeFieldRepr` trait which denotes the encoding of an element as a sequence of 64-bit words. Decoding a field element from octets involves going through instances of both traits, first the `PrimeFieldRepr`, then the `PrimeField` itself. Value validation is then split into two separate steps: the `PrimeFieldRepr` instance checks that the input has the expected number of bytes, but only `PrimeField` checks (as part of the `from_repr()` function) that the source value is lower than the modulus. In order to implement full little-endian convention, `sapling-crypto` relies on `PrimeFieldRepr` not validating the decoded integer against the field modulus. In that sense, the abstraction fails to abstract away encoding details.

In `sapling-crypto`, these encoding issues led to a mismatch between specification and implementation:

- Section 5.4.8.3 defines that $\text{abst}_{\mathbb{J}}$ maps a sequence of bits into a point on the Jubjub curve. It uses full little-endian convention to convert the 256-bit input into an integer z, such that $z \bmod 2^{255}$ will be the v coordinate of the point.
- Section 5.4.8.5 defines the first step of the "Group Hash" function as:
  $$P := \text{abst}_{\mathbb{J}}(\text{LEOS2IP}_{256}(\text{BLAKE2s-256}(D, \text{CRS} \parallel M))$$
  This definition is invalid, since LEOS2IP yields a big integer, while $\text{abst}_{\mathbb{J}}$ expects a sequence of bits. This seems to be a typographical error: LEOS2BSP should have been used. This may be a consequence of overzealous coalescing: $\text{abst}_{\mathbb{J}}$ starts with decoding bits into a big integer, and the combination of LEOS2BSP followed by that decoding into an integer is what LEOS2IP implements.
- Notwithstanding the typographical error above, the `group_hash()` function (in `src/group_hash.rs:18`) implements *big-endian* decoding, not little-endian. The BLAKE2 output is a sequence of octets; the leftmost (most significant) bit of the first octet is cleared, then the resulting sequence of bytes is interpreted as the v coordinate of the point with big-endian interpretation (`read_be()` is used). This sequence actually duplicates the `Point.read()` method in `src/jubjub/edwards.rs:95`, but without the full word-and-bits reversal, thereby using big-endian encoding throughout.

It shall be noted that CRH^{ivk} (section 5.4.1.5) is defined as using a BLAKE2 output, which formally consists of a sequence of octets (not a sequence of bits), and converting it into a big integer with LEOS2IP. Since the bit-swapping comes from the bit-level endianness convention, that specific function does not entail any bit-swapping. The implementation (see `src/primitives/mod.rs:89`) uses a `reverse()` call to reverse the order of octets within the array, then `read_be()`. It is possible that the intent of `group_hash()` was to follow the same sequence. However, `group_hash()` lacks the `reverse()` call.

Recommendation

Since the specification and the implementation of GroupHash do not match, one (or both) should be modified. As also noted, the specification is already formally invalid since it uses a big integer as input to $\text{abst}_{\mathbb{J}}$ instead of a bit string. Notwithstanding, the implementation uses big-endian encoding, which is opposite to the usual conventions in Zcash.

The use of little-endian ordering at the bit level is a source of confusion and implies cumbersome and expensive bit-swapping operations. NCC Group recommends that Zcash considers modifying the encoding conventions so that big-endian order is used for bits within octets (even if octets themselves use little-endian ordering within big integers). It should be possible

to avoid all bit-swapping operations. Moreover, in order to make the abstraction clearer, code from outside of the `pairing` library should refrain from manipulating the internal representation of a `PrimeFieldRepr` instance.

Finding	**Mempool Not Properly Cleared If Connecting or Disconnecting Tips Fails**
Risk	Informational Impact: Low, Exploitability: None
Identifier	NCC-Zcash2018-002
Status	Reported
Category	Other
Component	P2P Protocol
Location	`main.cpp`, line 2609
Impact	An edge condition during a reorg (blockchain reorganization in which the client discovers a new difficultywise-longest chain) may cause the client to behave differently than specified by the design, leaving both pre-activation and post-activation transactions inside the mempool. If this happens on a miner node, the miner may waste mining power on a block that includes transactions valid only by the pre-activation rules.
Description	The Overwinter Zcash release introduces a concept of activation height, which is a predetermined block height at which validation rules change. In particular, the block with `ACTIVATION_HEIGHT` height is the first block validated against the updated set of rules. As for the mempool, as stated in ZIP 200, "When the current chain tip height reaches `ACTIVATION_HEIGHT`, the node's local transaction memory pool SHOULD be cleared of transactions that will never be valid on the post-upgrade branch." See the PR that implements ZIP 200. This finding shows that edge cases exist in which both pre-activation and post-activation transactions may be left in the mempool. This may happen if the `ConnectTip` and `DisconnectTip` functions used to connect and disconnect new blocks to the active chain fail without setting the state to invalid. In particular, during a reorg, the tip of the chain is peeled off using the `DisconnectTip` function until the forking block is reached (see the `ActivateBestChainStep` function). On each invocation, the `DisconnectTip` function adds transactions to the memory pool, verifying them according to the ongoing height's validation rules. After all the blocks have been disconnected from the chain and the corresponding new blocks connected to the chain, the memory pool is finally cleared using the `mempool.removeWithoutBranchId` function called inside `ActivateBestChainStep`. As such, during a short period of time, both pre-activation and post-activation transactions are in the mempool, i.e., before they get cleared. If during this period, one of the `DisconnectTip` or `ConnectTip` calls return `false` , the client continues functioning with an inconsistent mempool. The `ConnectTip` function will return `false` (and not set the corresponding state flag to `MODE_INVALID`) in a case of a disk space error and `DisconnectTip` will return `false` in erroneous scenarios such as inconsistencies between block and undo data, read failures.
Recommendation	Whenever `DisconnectTip` or `ConnectTip` return `false`, clear the mempool of either pre-activation or post-activation transactions. The type of transactions to be cleared should be decided based on the active chain height. Even with the fix above, technically, the mempool will still contain both types of transactions over a short period of time in case of a reorg over activation height. If a more stringent property is to be enforced, by which pre- and post-activation transactions should never be in the mempool at the same time, then consider the following change: make the `Disconnect-`

`Tip` function activation height aware and delete post-activation transactions when switching to pre-activation block height. In order not to lose the deleted transactions, they need to be kept in an auxiliary store and considered for adding once traversing over the activation height with `ConnectTip`. The remaining question is when to clear the auxiliary store: it could be cleared on each `ActivationBestChainStep` exit.

Finding	**Private Key Decoding Has Limited Interoperability**
Risk	Informational Impact: Low, Exploitability: None
Identifier	NCC-Zcash2018-003
Status	Reported
Category	Uncategorized
Component	WalletDB
Location	`key.cpp`, function `ec_privkey_import_der()`
Impact	Some standard-compliant elliptic curve private key files will fail to be loaded.

Description

Elliptic curve private keys are meant to be encoded using ASN.1/DER structures that follow the syntax originally specified in SEC 1 (section C.4)[14]:

```
ECPrivateKey ::= SEQUENCE {
    version INTEGER { ecPrivkeyVer1(1) } (ecPrivkeyVer1),
    privateKey OCTET STRING,
    parameters [0] ECDomainParameters {{ SECGCurveNames }} OPTIONAL,
    publicKey [1] BIT STRING OPTIONAL
}
```

The `parameters` field is nominally optional, but is traditionally included in such structures. However, there are two possible formats for the `parameters`:

- the base field modulus, equation parameters, curve order, and conventional generator coordinates, may be encoded explicitly;
- for some specific curves of known parameter, a symbolic designation (`OBJECT IDENTIFIER`) may be used instead.

The use of a symbolic name yields a much shorter encoding. Moreover, this is the only format supported by PKIX for Internet-related standards,[15] and, as such, is more widely supported by existing cryptographic libraries.

The `ec_privkey_import_der()` function is a non-validating function that tries to locate the private key itself (`privateKey` field) and ignores the rest. It works by decoding the `SEQUENCE` tag and length, then skipping over the `version` field (after verifying its contents), and then grabbing the private key. The decoding process of the length of the `SEQUENCE` supports exactly two formats:

- "`81 xx`": length is encoded in byte `xx` (for lengths 128 to 255 bytes)
- "`82 xx yy`": length is encoded over two bytes `xx yy` (for lengths 256 to 65535 bytes)

However, this function does *not* tolerate the normal DER encoding when the `SEQUENCE` length is less than 128 bytes (such a short length is encoded over a single byte of value `0` to 127). In particular, if a private key for curve secp256k1 is encoded with its symbolic name (OID 1.3.132.0.10), then the whole contents of the `SEQUENCE` will use only 116 bytes, and the length will be encoded as a single byte of value 0x74. The `ec_privkey_import_der()` function cannot load such a private key.

In Zcash, private keys are decoded in two places:

[14] http://www.secg.org/sec1-v2.pdf
[15] https://tools.ietf.org/html/rfc5480, section 2.1.1

- in WalletDB, that handles encoding and decoding of private keys in a dedicated database;
- in the alert sending mechanism, the alert being signed with a private key hardcoded in the node at compilation time.

Since WalletDB typically generates key pairs itself, and the encoding function `ec_privkey_export_der()` uses the encoding format with explicit parameters, most serialized private keys used in practice have a format that `ec_privkey_import_der()` supports, which avoids the issue. As for alert signing, the private key is supposed to be generated by the installer using OpenSSL commands specified in the top comment in `sendalert.cpp`; in particular, the top command uses the "`-param_enc explicit`" argument, which instructs OpenSSL to use the explicit parameter encoding format. Therefore, common usage of the Zcash client is not impacted. However, interoperability with externally provided private keys is reduced, especially since the "explicit parameter" encoding format is discouraged in the PKIX world.

Recommendation

The length-decoding `ec_privkey_import_der()` function is currently implemented as follows:

```
    /* sequence length constructor */
    if (end - privkey < 1 || !(*privkey & 0x80u)) {
        return 0;
    }
    size_t lenb = *privkey & ~0x80u; privkey++;
    if (lenb < 1 || lenb > 2) {
        return 0;
    }
    if (end - privkey < lenb) {
        return 0;
    }
    /* sequence length */
    size_t len = privkey[lenb-1] | (lenb > 1 ? privkey[lenb-2] << 8 : 0u);
    privkey += lenb;
```

These lines may be replaced by the following:

```
    /* sequence length decoding */
    if (end - privkey < 1) {
        return 0;
    }
    size_t len = *privkey ++;
    if (len >= 0x80) {
        if ((size_t)(end - privkey) < (len - 0x80)) {
            return 0;
        }
        if (len == 0x81) {
            len = *privkey ++;
        } else if (len == 0x82) {
            len = *privkey ++;
            len = (len << 8) + *privkey ++;
        } else {
            return 0;
        }
    }
```

This modification would make `ec_privkey_import_der()` compatible with encoded private keys that use the "named curve" encoding format.

Finding	**Discrepancies Between Specification and Circuit Implementation**
Risk	Informational Impact: None, Exploitability: None
Identifier	NCC-Zcash2018-007
Status	Reported
Category	Other
Component	Sapling
Location	`sapling-crypto/src/circuit/sapling/mod.rs`, lines 296 to 308 `sapling-crypto/src/circuit/sapling/mod.rs`, lines 476 to 556
Impact	The discrepancies between the specification and the circuit implementation impair interoperability between Zcash implementations. Other independent implementations based on the protocol specification will be incompatible with the reference implementation.
Description	The order of arguments to NoteCommit$^{Sapling}_{rcm}$ in the Spend and Output circuits differs between the specification and the implementation found in *sapling-crypto*. According to the specification the note commitment (cm) is computed as follows[16]:

$$cm = \text{WindowedPedersenCommit}_{rcm}([1]^6 \,||\, g_d^\star \,||\, pk_d^\star \,||\, \text{I2LEBSP}_{64}(value))$$

However both Output and Spend circuits verify the note commitment as:

$$cm = \text{WindowedPedersenCommit}_{rcm}([1]^6 \,||\, \text{I2LEBSP}_{64}(value) \,||\, g_d^\star \,||\, pk_d^\star)$$

The relevant code in sapling-crypto for the Spend circuit can be seen below:

```
    // Place the value in the note
    note_contents.extend(value_bits);
}

// Place g_d in the note
note_contents.extend(
    g_d.repr(cs.namespace(|| "representation of g_d"))?
);

// Place pk_d in the note
note_contents.extend(
    pk_d.repr(cs.namespace(|| "representation of pk_d"))?
);
```

The implementation for the Output circuit is longer, but similarly injects the $value$ before $g_d^\star$ and $pk_d^\star$, contrary to the specification.

Recommendation	Both parameter orders are appropriate from a security point of view, since the elements have a fixed length, thus making the concatenation unambiguous. However, either the specification or the implementation should be modified so that they match.

[16] Zcash Specification, version 2018.0-beta-19, Section 5.4.7.2 (p. 57, Windowed Pedersen Commitments)

Finding	**Typographical Inconsistencies in Zcash Specification**
Risk	Informational Impact: None, Exploitability: None
Identifier	NCC-Zcash2018-008
Status	Reported
Category	Other
Component	Sapling
Location	Sapling
Impact	Discrepancies between the specification and the implementation make protocol evolution and auditing more difficult.
Description	Listed here are a few discrepancies between the specification and the implementation, that are mere typographical errors, and should be fixed in the specification for the sake of consistency and clarity of documentation.

Page 20: A Signature with Re-Randomizable Keys is defined as using two algorithms:
Sig.RandomizePrivate : Sig.Random × Sig.Private → Sig.Private
Sig.RandomizePublic : Sig.Random × Sig.Public → Sig.Public
However the concrete instantiation RedDSA (page 55), is defined by algorithms:
RedDSA.RandomizePrivate : RedDSA.Private × RedDSA.Random → RedDSA.Private
RedDSA.RandomizePublic : RedDSA.Public × RedDSA.Random → RedDSA.Public
The random and public/private key arguments are thus in reverse order. The meaning will be clear to most readers, but consider permuting the arguments in RedDSA to clarify.

Page 49: The Pedersen hash is defined as a linear combination computed over a sequence of generator points $\mathcal{I}_i^D$. The input data is split into segments of c chunks of 3 bits each; each segment is encoded into a scalar by which a dedicated generator point is multiplied. Security relies on the generators being all different. However, the formula that defines $\mathcal{I}_i^D$ uses floor($\frac{i-1}{c}$), which makes generators for $i = 1$ to c identical to each other. This is obviously a confusion between the segment index and the chunk index: the formula for $\mathcal{I}_i^D$ expects the chunk index, but it is used with the segment index. The implementation uses the segment index, which is the proper thing to do.

Page 55: Typo in definition of `RedDSA.RandomizePublic`: the type signature of `RedDSA.RandomizePublic` is followed by a function called `RedDSA.RandomizePrivate`.

Page 56: The Spend Authorization Signature is defined as using RedJubjub, parametrized with a generator point $\mathcal{G}$, defined in section 4.2.2. However, $\mathcal{G}$ is not defined in section 4.2.2, nor anywhere else in the specification document. The implementation uses (in `sapling_crypto`, file `src/jubjub/mod.rs`, line 281):
$\mathcal{G} = \text{FindGroupHash}^{\mathbb{J}}(\text{"Zcash\_G\_"}, \text{""})$

Finding	**Bias in Random Field Element Generators for Underlying Representation**
Risk	Informational Impact: None, Exploitability: None
Identifier	NCC-Zcash2018-011
Status	Reported
Category	Cryptography
Component	Sapling
Location	https://github.com/zcash-hackworks/sapling-crypto/blob/5687acfaf83438a993fccc14ab487b67e4afbc68/src/jubjub/fs.rs#L46
Impact	If this randomness function is used to generate secrets in the future, an attacker will have an advantage in predicting them.
Description	The Sapling cryptography library implements elements of the scalar field of the Jubjub curve in two layers. At the higher level is the `Fs` type, which offers a clean interface into these elements, and at the level below is the `FsRepr` type, which handles the necessary lower-level functionality for these elements. Currently, an `FsRepr` is implemented as an array of four `u64` values. The `FsRepr` type's implementation of the `Rand` trait leverages `rng.gen()` to implement the required `rand()` function. This means that the random generator will create a random array of four `u64` values. However, this means that randomly generated `FsRepr`s are not necessarily elements of `Fs`. This is mitigated by the appropriate checks in the higher level `Fs` type's `rand` implementation, and so is not a risk to the *current* implementation. This fact is reflected in the "Informational" level of risk in this finding. Nevertheless, the `FsRepr` trait is public, which means that future development may assume this function will always offer a valid element of the scalar field.
Recommendation	Any trait implemented by `FsRepr` that is likely to change across implementations should be made private, then offered transparently through an equivalent trait implementation in `Fs` (i.e. a trait implementation that calls the equivalent from `FsRepr` along with any changes necessary to offer it publicly). Additionally, any methods that could potentially be unsafe should be explicitly labelled as such.

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 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.