NU4 Cryptographic Specification and Implementation Review

Zcash

September 3, 2020 – Version 2.0

Prepared for
Benjamin Winston

Prepared by
Paul Bottinelli
Marie-Sarah Lacharité
Thomas Pornin

Executive Summary

Synopsis

In June 2020, the Electric Coin Company engaged NCC Group to conduct a security review of the six Zcash Improvement Proposals (ZIPs) that constitute the core of the upcoming "Canopy" upgrade (also called "NU4") to the Zcash network. This upgrade coincides with the first Zcash halving and will initiate a new development fund for the next four years. The audit was meant to identify vulnerabilities that may result from application of the ZIPs, including consensus breaches induced by diverging implementations arising from incomplete or unclear specifications. NCC Group assigned three consultants to the specification audit for a total of eight person-days.

In a continuation of this effort, NCC Group was tasked with reviewing the implementation of the six aforementioned ZIPs in August 2020. The audit was meant to assess the consistency of the implementation with the proposed changes in the protocol specification, identify vulnerabilities that may have been introduced, as well as review general programming practices. NCC Group assigned two consultants to the implementation audit for a total of eight person-days.

Scope

The audit scope was the six following ZIPs:

- ZIP 214 – Consensus rules for a Zcash Development Fund: https://zips.z.cash/zip-0214
- ZIP 207 – Funding Streams: https://zips.z.cash/zip-0207
- ZIP 251 – Deployment of the Canopy Network Upgrade: https://zips.z.cash/zip-0251
- ZIP 211 – Disabling Addition of New Value to the Sprout Value Pool: https://zips.z.cash/zip-0211
- ZIP 212 – Allow Recipient to Derive Sapling Ephemeral Secret from Note Plaintext: https://zips.z.cash/zip-0212
- ZIP 215 – Explicitly Defining and Modifying Ed25519 Validation Rules: https://zips.z.cash/zip-0215

The ZIPs' published states as of June 5th, 2020 were used for the first part of the audit. Their modification history is available in the GitHub repository at https://github.com/zcash/zips.

In the second part of the audit, the implementation review was centered around version 3.1.0[1] of the Zcash project, which included the implementation of the ZIPs described above, in the form of the following Pull Requests (PRs):

- ZIP 207 and ZIP 214: https://github.com/zcash/zcash/pull/4560
- ZIP 251: https://github.com/zcash/zcash/pull/4487
- ZIP 211: https://github.com/zcash/zcash/pull/4489
- ZIP 212: https://github.com/zcash/zcash/pull/4578 and https://github.com/zcash/librustzcash/pull/258
- ZIP 215: https://github.com/zcash/zcash/pull/4581
- Changes to the ed25519-zebra library to implement the validation rules of ZIP 215: https://github.com/ZcashFoundation/ed25519-zebra/pull/24

Finally, two versions of the Zcash Protocol Specification, dating from August 3rd, 2020 (version 2020.1.12[2]) and August 11th, 2020 (version 2020.1.13[3]), which included modifications introduced by the above ZIPs, were also used as a references for phase 2.

Appendix A on page 30 summarizes all the relevant references pertaining to the review.

Limitations

One of the goals of a security review of specification documents is to identify points that are not covered in sufficient clarity, and may thus result in incompatibilities when implemented by separate teams. The notion of clarity is subjective and relies on assumptions about the level of understanding and competence of future implementers. Moreover, in the first phase of the assessment, which took place in the middle of June 2020, the ZIPs themselves were not the final form of the proposed changes.

Thus, phase 2 of the audit, which was primarily focused on the implementation of the proposed changes, also included some limited review of the updated ZIPs and protocol specification. These modifications were introduced between the two phases of the assessment, partly as part of the standard Zcash development process, but also to address some of the points raised during phase 1.

NCC Group tried to err on the side of caution, and many of the remarks in this report may correspondingly look like minor quibbles.

[1] https://github.com/zcash/zcash/tree/v3.1.0
[2] https://github.com/zcash/zips/blob/fb64b2e4303b332ef8960fc6bbf34b0598596c5d/protocol/protocol.pdf
[3] https://github.com/zcash/zips/blob/6e5278ed95e334dc861838da26857d1c0dcf638f/protocol/protocol.pdf

Key Findings

No potentially serious vulnerability was found in the reviewed ZIPs themselves.

While reviewing ZIP 215, NCC Group noticed that the current implementation of Ed25519 signature verification, used in JoinSplit descriptions, did not faithfully adhere to the current protocol in some specific edge cases. Though none of these cases will ever be hit by non-malicious implementations, they could still be used to induce a breach of consensus between the current Zcash implementation and theoretical alternate implementations that would closely adhere to the published protocol. Application of ZIP 215 will indirectly mitigate *most* of this issue, but an edge case will remain possible, unless explicitly addressed. This issue is detailed in finding NCC-ZCHX006-001 on page 23.

This edge case was subsequently addressed in the latest version of the protocol specification.

Strategic Recommendations

In order to maintain strict consistency in wording and typography across future ZIPs, an explicit and detailed style guide should be written and maintained.

The code base could also benefit from more specific and detailed comments, particularly since the constant updates to the code in the form of ZIPs add non-trivial layers of complexity.

Dashboard

Target Metadata

Name	Canopy Network Upgrade
Type	Protocol Specification and Implementation
Platforms	Zcash

Engagement Data

Type	Specification and Implementation Review
Method	Architectural and Code Review
Dates	2020-06-01 to 2020-08-14
Consultants	3
Level of Effort	16 person-days

Targets

ZIP 214	https://zips.z.cash/zip-0214
ZIP 207	https://zips.z.cash/zip-0207
ZIP 251	https://zips.z.cash/zip-0251
ZIP 211	https://zips.z.cash/zip-0211
ZIP 212	https://zips.z.cash/zip-0212
ZIP 215	https://zips.z.cash/zip-0215
Implementation of ZIP 207 and ZIP 214	https://github.com/zcash/zcash/pull/4560
Implementation of ZIP 251	https://github.com/zcash/zcash/pull/4487
Implementation of ZIP 211	https://github.com/zcash/zcash/pull/4489
Implementation of ZIP 212	https://github.com/zcash/zcash/pull/4578
Changes to librustzcash for ZIP 212	https://github.com/zcash/librustzcash/pull/258
Implementation of ZIP 215	https://github.com/zcash/zcash/pull/4581
Changes to ed25519-zebra for ZIP 215	https://github.com/ZcashFoundation/ed25519-zebra/pull/24

Finding Breakdown

Critical issues	0
High issues	0
Medium issues	0
Low issues	2
Informational issues	0
Total issues	**2**

Category Breakdown

Cryptography	1
Data Validation	1

Component Breakdown

Ed25519	1
Wallet	1

Key

Critical | High | Medium | Low | Informational

Audit Scope

The audit covers the six ZIPs described in the Scope section of the Executive Summary on page 2. Additionally, Appendix A on page 30 summarizes all the relevant references pertaining to the review.

All ZIPs were reviewed in their published state, as of June 5th, 2020. In the following sections, NCC Group first summarizes each ZIP and offers remarks, most of which are editorial in nature. A later section lists more editorial remarks that apply to several ZIPs, in particular, wording consistency issues.

In general, no security vulnerability was found in the audited ZIPs. A potential consensus breach issue was detected in the current implementation during the analysis of ZIP 215; this is not a flaw of ZIP 215 itself, whose application will, in fact, mitigate most of the issue. This is detailed in the text below, and in finding NCC-ZCHX006-001 on page 23.

Individual ZIP Reviews

This section includes NCC Group's remarks for each of the six ZIPs that were in scope.

ZIP 214

ZIP 214 describes rule changes interpreting the new Zcash Development Fund structure as proposed in ZIP 1014.[4] More specifically, it specifies the introduction of a Development Fund corresponding to 20% of the block subsidy, split into the following three funding streams:

- FS_ECC: 35% for the Electric Coin Company (corresponding to 7% of the total block subsidy);
- FS_ZF: 25% for Zcash Foundation (5% of the total block subsidy);
- FS_MG: 40% for additional "Major Grants" for large-scale long-term projects (8% of the total block subsidy).

These changes will be activated in Canopy, corresponding to a block height of 1046400.

This ZIP is semantically consistent with the stated goal and the other ZIPs. The following remarks can be made:

- In the *Specification* section, the activation height of Canopy on Testnet is still undefined, and thus is replaced with the placeholder string "xxxxxxx". Other not-yet-defined elements in the ZIPs use the "TODO" placeholder. By using a non-standard placeholder, this item may evade ulterior checks for consistency and completeness of the information in the ZIPs.
- The following sentence is not entirely clear:

  > In this case the total amount of funding streams assigned to direct grantees MUST be subtracted from the funding stream for the remaining MG slice

  Namely, the usage of the word *amount* when referring to streams is ambiguous. When referring to funding streams, the word *number* should be used since in this case stream is a countable noun. However, the intention of using the word *amount* might be to refer to the *value*. But in this case, it is unclear how an *amount of funding streams* can be *subtracted from the funding stream*. In that case, it might be clearer to say something along the lines of *subtracted from the value* of *the funding stream for the remaining MG slice*.
- In this sentence:

  > For each network upgrade after Canopy requiring modifications to the set of direct grantees, a separate ZIP would be published specifying those modifications.

  The term "would" is not one of the RFC 2119 key words, and it is unclear whether this sentence defines a formal requirement, or is a vague statement of intent. NCC Group recommends amending this sentence with an explicit "MUST", "MAY" or "SHOULD", depending on the intent.
- In the *Specification* section, the sentence:

[4] https://zips.z.cash/zip-1014

> The funding streams are defined for Testnet are identical

contains one "are" too many and should be rephrased.

ZIP 207

ZIP 207 specifies the procedure by which the funding streams are used to implement the Zcash Development Fund. It describes the precise mechanisms to implement the funding streams as defined in ZIP 214 and motivated by ZIP 1014. As a security measure, each stream specifies a number of different addresses. The recipient address for a given funding stream is then updated about once per month, resulting in 48 different periods during the total duration of the Zcash Development Fund lifespan (four years).

Note that parts of ZIP 207 were audited by NCC Group in a previous engagement[5]; ZIP 207 was initially scheduled for inclusion in the Blossom network upgrade, but was then delayed and rescheduled.

NCC Group verified that all formulas and rules specified in ZIP 207 are consistent with the stated goals and the other ZIPs. In particular, given a start height equal to the Canopy update and an end height corresponding to the next halving (at block height 2726400), all 48 addresses at indices $0 \ldots 47$ are indeed obtained an equal number of times (namely 35,000) by the $\mathsf{AddressIndex}(\mathsf{height})$ function; moreover, the claim that:

> all active funding streams change the address they are using on the same block height schedule

was also independently verified and holds within the given start and end heights. Namely, even with funding streams starting later than the Canopy activation height, address changes are performed on the same block height schedule.

Some extra remarks:

- In the *Definitions* section, a reference is made to sections 5.3, 7.7, and 7.8 of the Zcash protocol specification for the definition of some constants and functions; however, the constant $\mathsf{PostBlossomHalvingInterval}$ is not defined in any of these sections (it is defined in section 7.6.3 of the protocol specification).
- In the *Funding Streams* section, the computation of the $\mathsf{Address}$ at a given height is performed by the following computation:

$$\mathsf{Address}(\mathsf{height}) = \mathsf{FundingStream}[\mathsf{FUND}].\mathsf{Addresses}[\mathsf{FundingStream}[\mathsf{FUND}].\mathsf{AddressIndex}(\mathsf{height})]$$

  However, this value is associated with a given $\mathsf{FundingStream}$. Thus, it probably should be described as follows:

$$\mathsf{FundingStream}[\mathsf{FUND}].\mathsf{Address}(\mathsf{height}) = \mathsf{FundingStream}[\mathsf{FUND}].\mathsf{Addresses}[\mathsf{FundingStream}[\mathsf{FUND}].\mathsf{AddressIndex}(\mathsf{height})]$$

- The *Example implementation* is known to be incomplete and not up-to-date; it was agreed that it would not be re-evaluated during this audit.

ZIP 251

ZIP 251 defines the deployment of the Network Upgrade 4 (NU4), also known as the Canopy network upgrade. Since all major changes are described in their own ZIPs, ZIP 251 is a straightforward list of references to the relevant ZIPs, along with exact rules for backward compatibility and handling of existing and new connections during and after the transition to Canopy.

Reference `[5]` to ZIP 207 was incorrectly pointing to ZIP 208, but this was independently fixed by Zcash concurrently with the audit, on June 9th, 2020.

[5] https://www.nccgroup.com/us/our-research/zcash-blossom-specification-report/

ZIP 211

ZIP 211 is a step toward deprecation and ultimately removal of the support for Sprout shielded transactions; it removes the ability to add new value to the Sprout value pool balance.

The proposed modification is straightforward and does not induce any security issue, since Sapling shielded transactions can handle all the functionalities of Sprout shielded transactions, and a migration tool that can move funds from Sprout to Sapling is already deployed.

NCC Group offers the following remarks:

- In the *Terminology* section, "Sapling" is defined (by way of ZIP 205), but not "Sprout", only "Sprout value pool balance".

  "Sprout value pool balance" is defined by reference to ZIP 209, which says its definition (the sum of all transactions' `vpub_old` fields minus the sum of all `vpub_new`) is implied by section 4.11 of the Zcash protocol specification. The implication could be made more explicit: (1) the section refers only to the transparent value pool, (2) it considers pools only at the transaction and block levels, and (3) a pool's balance is never defined, nor does "balance" appear anywhere but the section title.

  NCC Group recommends to update section 4.11 of the Zcash protocol specification to include an explicit mention of the Sprout value pool and its balance, whether at the transaction or block level. (The Sprout value pool does not appear to be explicitly mentioned at all in the protocol specification document.) Consider also mentioning here that the concept of balance can be extended in the natural way to the block chain level.

- In the *Abstract* section, the sentence "add new value to the Sprout value pool balance" is arguably redundant and could be shortened into "increase the Sprout value pool balance" or "add new value to the Sprout value pool".

- In the *Motivation* section, the term "Sprout" in the sentence "the use of Sapling shielded transactions will replace the use of Sprout" may be more precisely written as "Sprout shielded transactions".

  In the sentence "the Zcash specification and implementation incurs complexity", the verb should be spelled "incur".

- In the *Specification* section, there are two MUST-qualified requirements:

  - the `vpub_old` field of each JoinSplit description MUST be `0`;
  - nodes and wallets MUST disable sending to Sprout addresses.

  The first requirement is clear and coincides with the ZIP's title. The second, however, appears to additionally forbid fully shielded Sprout transactions that would not change the value pool balance. NCC Group recommends that the second MUST statement be clarified. If forbidding fully shielded Sprout transactions is not the intended effect, then the second statement should be augmented with the qualifier "from transparent addresses". If it is, however, then consider renaming the ZIP to reflect that it will disable all transfers to Sprout shielded addresses.

- In the *Rationale* section, it is asserted that "the code changes needed are very small and simple, and their security is easy to analyse". This is true in the context of a system (node or client) that already supports Sapling shielded transactions; but, in all generality, removing support of Sprout shielded transactions requires supporting Sapling shielded transactions, and Sprout support cannot be removed right away precisely because it cannot be assumed that all clients have Sapling support. The proposed change is one step in a long-term removal process whose support is not small and simple.

ZIP 212

ZIP 212 suggests a modification to the contents of encrypted notes so that privacy properties of Sapling shielded transactions no longer depend on the soundness of the underlying zk-SNARK.

In the current protocol, an encrypted note contains a value called **rcm** that is generated randomly and uniformly modulo the prime r, which is the order of the largest sub-group of prime order in the Jubjub curve. The note itself is

symmetrically encrypted using a key derived from an ECDH key exchange over the Jubjub curve; the sender generates an ephemeral ECDH key pair, the private key **esk** being chosen randomly and uniformly among the possible private keys, i.e. non-zero integers modulo r.

The proposed change replaces the generation of **rcm** and **esk** with the generation of a single 32-byte random seed **rseed** from which **rcm** and **esk** are both derived with a deterministic PRF. In the note plaintext format, **rseed** replaces **rcm**.

All previous functionality is maintained, since the receiver can recompute **rcm** from **rseed** by using the PRF. The receiver can additionally *verify* that the value of the ephemeral ECDH public key used for the encryption corresponds to the value of **esk** that can be similarly recomputed from **rseed**. In a strict sense, the use of the private key **esk** for ECDH *and* as part of the encrypted plaintext might theoretically induce a weakness in the encryption system, but this is very improbable in practice, since the ECDH key and the note plaintext are "separated" from each other by the key expansion PRF and the symmetric encryption layer. An unwanted interaction between the value of **esk** in the plaintext and the ECDH key exchange would require contrived definitions of both the PRF and the symmetric encryption algorithm. NCC Group deems such a case implausible in practice.

Some extra remarks:

- In the *Terminology* section, the **ToScalar** function is said to be defined in section 4.4.2 of the Zcash protocol specification; but there is no such section. **ToScalar** is defined in the section 4.2.2 (although already named in the section 4.1.2).
- The used PRF (called $\mathsf{PRF}^{\mathsf{expand}}$ in the Zcash protocol specification) uses an extra diversifier argument; in the proposed ZIP 212, that argument is a byte of value 4 (for **rcm**) or 5 (for **esk**). It is important for the security analysis that such diversifier bytes are distinct for all different uses of the PRF in the protocol. This is the case here (the Zcash protocol specification uses only values 0 to 3, and ZIP 32[6] uses values 16 to 22, and 128). It may be worth maintaining a formal list of used diversifier values so that unwanted reuse is avoided in future protocol evolutions.
- In the current protocol, **esk** is chosen uniformly among non-zero integers modulo r. In the proposed change, **esk** is the result of a call to **ToScalar**, which is the reduction of a large integer modulo r; thus, the value $\mathsf{esk} = 0$ now becomes possible. It is, however, very improbable, since $r \approx 2^{251.86}$. Since the PRF is based on the BLAKE2b hash function, finding an **rseed** value that yields $\mathsf{esk} = 0$ would require a partial preimage attack on BLAKE2b, and it can be reasonably postulated that no such attack exists with cost less than $2^{251.86}$ operations on average. This cost is sufficiently high that the possibility can be ignored in practice. Moreover, since operations on Jubjub use complete formulas, a zero ECDH key would not actually be mathematically incorrect.

ZIP 215

ZIP 215 proposes to remove restrictions on low-order points in Ed25519 public keys and signatures. The current protocol includes some historical restrictions inherited from an old version of libsodium, which make some points in signatures and public keys unconditionally rejected.

During the analysis of ZIP 215, it was noticed that the current standard implementation of Zcash[7] does not strictly conform to the Zcash protocol specification: some points that should be rejected as per the protocol are accepted by the implementation and vice versa. This does not seem to be exploitable for any kind of signature forgery, but it could lead to consensus breaches between the Zcash implementation and nodes using another implementation that would faithfully adhere to the protocol. Such an alternate implementation does not currently exist, making the issue purely theoretical at the moment. This is detailed in finding NCC-ZCHX006-001 on page 23.

Application of ZIP 215 will remove *most* of these issues, except that it does not address the edge case of an invalidly encoded point with implied coordinate $x = 0$ but with an expected least significant bit of value 1. Such nominally invalid encodings are currently accepted by libsodium and thus the Zcash implementation, but they are explicitly rejected by

[6] https://zips.z.cash/zip-0032
[7] https://github.com/zcash/zcash

RFC 8032,[8] to which ZIP 215 refers. In order to avoid further potential consensus breaches due to incompatible handling of such cases, NCC Group **recommends** that ZIP 215 be amended to explicitly specify the expected implementation behavior when such a point is encountered as a public key or as part of a signature.

As was noted in ZIP 215, accepting low-order points does not endanger the security properties expected of signatures, since signatures leveraging such points require invalidly generated public keys.

Editorial and Typography Remarks

This section includes NCC Group's remarks that apply to all ZIPs, in particular, typographical consistency issues.

Links to the Protocol Specification

The ZIPs refer to various concepts and items defined in the Zcash Protocol Specification; in most cases, a specific section number is given, and the protocol specification itself is identified by "version 2020.1.1 *or later*". Generally speaking, section numbering may change in future versions of the protocol specification, if sections are added (this has not yet happened thanks to a conscious effort from the protocol document editors, but it may occur in the future). Any such renumbering may make references by section numbers incorrect.

This is mitigated in practice by the use of *anchored links*, in which the HTML link itself contains a symbolic identifier that points at the specific section within the protocol document. Symbolic anchors are resilient to renumbering. However, anchorless links are used in the following references:

- In ZIP 214, *Terminology* section, for the definition of the terms "block subsidy" and "halving" (the text explicitly mentions sections 3.9 and 7.7).
- In ZIP 251, *Backward compatibility* section, for the version group ID (the text refers to section 7.1).
- In ZIP 212, *Terminology* section, for the definition of the function ToScalar (the text wrongly mentions section 4.4.2, but there is no such section in the protocol specification; the ToScalar function is defined in section 4.2.2 of the protocol).
- In ZIP 215, *Motivation* and *Specification* sections, for the JoinSplitSig validation rules (there is no mention of a section in the reference from the *Motivation* section; in the *Specification* section, the text points to section 5.4.5 of the protocol).

Recommendation: Anchored links should be used systematically in order to resist potential section renumbering in future versions of the protocol specification. Such links also improve the reading experience, since clicking them brings the reader to the right section of the protocol.

Standard Key Words for Requirement Levels

All of the ZIPs, in their respective *Terminology* section, refer to RFC 2119[9] for the interpretation of some standard key words such as "MUST" or "SHOULD" that designate requirement levels. The ZIPs, however, differ in their wording:

> The key words "MUST", "SHOULD", "SHOULD NOT", and "MAY" in this document are to be interpreted as described in RFC 2119.
>
> The key words "MUST", "SHALL", and "SHOULD" in this document are to be interpreted as described in RFC 2119.
>
> The key words "MUST", "MUST NOT", "SHOULD", and "MAY" in this document are to be interpreted as described in RFC 2119.
>
> The key words "MUST", "SHOULD", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119.

[8] https://www.rfc-editor.org/rfc/rfc8032.html
[9] https://www.rfc-editor.org/rfc/rfc2119.html

> The key words “MUST” and “MUST NOT” in this document are to be interpreted as described in RFC 2119.
>
> The key words “MUST” and “MUST NOT” in this document is to be interpreted as described in RFC 2119.

Note, in particular, that ZIP 214’s wording does not include the key word “MAY”, which is nonetheless used (in uppercase) in the text of the ZIP (in the *Dev Fund Recipient Addresses*).

Recommendation: Harmonize the wording across all ZIPs. RFC 2119 itself suggests an all-purpose wording that includes *all* the key words described by RFC 2119:

> The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.

Network Naming

The production and test networks are designated by the terms “Mainnet” and “Testnet”, respectively. These terms appear in ZIPs 207, 214 and 251; in ZIPs 207 and 214, they are capitalized systematically, but in ZIP 251, they start with a lowercase letter when not used at the start of a sentence. In the Zcash protocol specification, “mainnet” does not appear, but “testnet” is used, with a lowercase “t”, like in ZIP 251.

It shall also be noted that ZIPs 207, 214 and 251 refer to the test network and production networks “as defined in the Zcash Protocol Specification”, but the Zcash protocol specification does not actually define these expressions. The protocol *uses* the expressions “production network” and “test network” quite liberally, but they don’t seem to be formally defined anywhere in the document.

Recommendation: The production and test networks should be properly defined and named in the protocol specification. Consistent capitalization rules should then be applied to the chosen names.

Audit Scope

The scope of the second phase of the audit covers the implementation of the six pertinent ZIPs, which is part of the v3.1.0 release,[10] as well as changes to `librustzcash` and to the `ed25519-zebra` library to fix some discrepancies previously identified.

Appendix A on page 30 summarizes all the relevant references pertaining to the review.

Some specific focus areas were identified by the Zcash team prior to performing the implementation review, including:

- The implementation of the funding streams logic and height computations, as part of the changes introduced by ZIP 207 and ZIP 214;
- Recent changes to ZIP 212 and its implementation, particularly with the introduction of a grace period to facilitate the transition to the modifications required by ZIP 212;
- General discrepancies between specification and implementation which might result in consensus breaches.

The remainder of this section is organized in a similar fashion as the Audit Notes of Phase 1, namely some individual ZIP comments followed by more general considerations applicable to the code base.

Individual ZIP Reviews

In the following subsections, we list considerations pertaining to the changes introduced by the implementation of the different ZIPs. These observations do not have direct security impacts, but indicate slight differences between specification and implementation that may be worthwhile to understand and/or resolve.

ZIP 207 and ZIP 214

ZIP 207 and ZIP 214 introduce changes supporting the introduction of the new Zcash Development Fund structure as proposed in ZIP 1014. When reviewing the changes introduced by these ZIPs, particular care was given to the mathematical correctness of the functions used to compute the value of funds and the recipient address at a specific block height. Some specific discussions are provided below.

- Some discrepancies exist between the protocol specification definition of the quantity `HeightForHalving`, its implementation in the form of the `HalvingHeight()` function, and some comments in the code. The protocol specification defines the `HeightForHalving` function as the following:

$$\mathsf{HeightForHalving}(\mathsf{halving} : \mathbb{N}) := \min(\{\mathsf{height} : \mathbb{N} \mid \mathsf{Halving}(\mathsf{height}) = \mathsf{halving}\})$$

  In comparison, the implementation defines it as:

$$\mathsf{HalvingHeight}(i) := \max(\{\mathsf{height} : \mathbb{N} \mid \mathsf{Halving}(\mathsf{height}) < i\}) + 1$$

  More specifically, the file `src/consensus/params.cpp` states the following:

```
// HalvingHeight(i) := max({ height : N | Halving(height) < i }) + 1
//
// Halving(h) returns the halving index at the specified height.  It is
// defined as floor(f(h)) where f is a strictly increasing rational
// function, so it's sufficient to solve for f(height) = halvingIndex
// in the rationals and then take ceiling(height).
```

  This is correct as long as the `Halving()` function is monotonically increasing, or a *strictly increasing rational function*, as the comment states. This difference does not have an impact and probably never will, since halving is never expected to be reverted.

[10] https://github.com/zcash/zcash/tree/v3.1.0

However, there is a difference in the domains of these two functions, namely between its formal definition and implementation.

More specifically, the definition of `HeightForHalving` in the protocol specification requires a single argument, a *positive* integer for which the corresponding halving block height is expected. In comparison, the implementation requires two *signed* integer arguments `int Params::HalvingHeight(int nHeight, int halvingIndex)`. This might lead to potential misuses of this function in the following ways:

1. `HalvingHeight()` could be called with a negative `halvingIndex` (or a negative `height`), resulting in negative return values.
2. `HalvingHeight()` returns wrong results when the height passed in as parameter is inconsistent with the `halving` factor. Namely, the values returned by the `Halving()` function are increased at heights `1046400`, `2726400`, `4406400` for halving values of `1`, `2`, `3` (which was also verified by an independent naive implementation).

As a result, the function `HalvingHeight()` should return these values when called with a `halvingIndex` of `1`, `2`, `3` respectively. This is indeed the case when `HalvingHeight()` is called with an `nHeight` argument larger than the Blossom activation height. However, this is not the case when the height is smaller than the Blossom activation height. A transcript for the two cases with values of `halvingIndex` between 1 and 3 is provided below.

```
HalvingHeight(height, i), height >= blossomActivationHeight, i = [1..3]
---
HalvingIndex: 1, Height: 1046400
HalvingIndex: 2, Height: 2726400
HalvingIndex: 3, Height: 4406400

HalvingHeight(height, i), height < blossomActivationHeight, i = [1..3]
---
HalvingIndex: 1, Height: 850000
HalvingIndex: 2, Height: 1690000
HalvingIndex: 3, Height: 2530000
```

While this behavior is to be expected to some extent, and abuses of this function seem unlikely given its current use in the code base, consider more closely following the definition of the protocol specification in order to completely avoid misuses.

Another impact of this discrepancy affects the value of `FoundersRewardLastBlockHeight`, which is defined in the protocol specification as

$$\mathsf{FoundersRewardLastBlockHeight} := \max(\{\mathsf{height} : \mathbb{N} \mid \mathsf{Halving}(\mathsf{height}) < 1\})$$

But is implemented as follows:

```
int Params::GetLastFoundersRewardBlockHeight(int nHeight) const {
    return HalvingHeight(nHeight, 1) - 1;
}
```

Similar to the previous consideration related to the `HalvingHeight`, the result is dependent on the height provided as an argument, and this function will either correctly return `1046399`, or `849999` if the height is smaller than the Blossom activation height.

- In the protocol specification, the value of a Funding Stream is defined as:

$$\text{fs.Value(height)} := \begin{cases} 0, & \text{if height} < \text{CanopyActivationHeight} \\ \text{floorBlockSubsidy(height)} \cdot \frac{\text{fs.Numerator}}{\text{fs.Denominator}}, & \text{if fs.StartHeight} \leq \text{height and height} < \text{fs.EndHeight} \\ 0, & \text{otherwise} \end{cases}$$

In comparison, in `src/consensus/funding.cpp`, the `Value` computation itself returns a non-zero value, regardless of the block height.

```
CAmount FSInfo::Value(CAmount blockSubsidy) const
{
    // Integer division is floor division for nonnegative integers in C++
    return CAmount((blockSubsidy * valueNumerator) / valueDenominator);
}
```

In practice, this does not seem to have an impact. Indeed, in both places where it is used, namely in the functions `GetActiveFundingStreamElements()` of the same file and `getblocksubsidy()` in `src/rpc/mining.cpp`, appropriate bound checking is in place. The value is computed only if the Canopy upgrade is active and the current height is within the Funding Stream start and end heights. The code excerpted below showcases this behavior.

```
std::set<FundingStreamElement> GetActiveFundingStreamElements(
...
if (fs && nHeight >= fs.get().GetStartHeight() && nHeight < fs.get().GetEndHeight()) {
    requiredElements.insert(std::make_pair(
        fs.get().RecipientAddress(params, nHeight),
        FundingStreamInfo[idx].Value(blockSubsidy)));
}
```

```
UniValue getblocksubsidy(const UniValue& params, bool fHelp)
{
...
    if (canopyActive) {
        UniValue fundingstreams(UniValue::VARR);
        auto fsinfos = Consensus::GetActiveFundingStreams(nHeight, consensus);
        for (auto fsinfo : fsinfos) {
            CAmount nStreamAmount = fsinfo.Value(nBlockSubsidy);
            ...
```

However, these checks are performed at varying levels of depth in the codebase. As such, they might be missed by developers unacquainted with the code.

More specifically, the function `getblocksubsidy()` first checks that Canopy is active and then obtains the active Funding Streams with `GetActiveFundingStreams()` which ensures that the Streams are active at the current height. It then calls the `Value()` function.

In comparison, the function `GetActiveFundingStreamElements()` iterates over the Funding Streams, checks they are active, and then calls the `Value()` function. The checks that Canopy is active are performed one level above, namely in the `ContextualCheckTransaction()` and `SetFoundersRewardAndGetMinerValue()` functions, where calls to `GetActiveFundingStreamElements()` are properly guarded.

In essence, the core function defined in the protocol documentation does some explicit bound checking, while this is performed in other functions in the implementation. Consider properly documenting at what level these checks are performed.

- The protocol specification defines the quantity `PostBlossomHalvingInterval` as

$$\text{PostBlossomHalvingInterval} := \text{floor}(\text{PreBlossomHalvingInterval} \cdot \text{BlossomPoWTargetSpacingRatio})$$

In the code, more specifically in `src/consensus/params.h`, this quantity is defined as follows:

```
#define POST_BLOSSOM_HALVING_INTERVAL(preBlossomInterval) \
    (preBlossomInterval * Consensus::BLOSSOM_POW_TARGET_SPACING_RATIO)
```

Namely, the floor is omitted and the function is defined as a macro with the `preBlossomInterval` argument. It seems like this is done in order to better support testing by providing different values for the `preBlossomInterval` .

However, this quantity is constant for a given set of parameters. Defining it as a macro might lead to unexpected misuses compared to defining a non-variable value for `PostBlossomHalvingInterval` and a different one to be used solely for testing purposes.

- In `src/main.cpp`, an `if` statement was added on line `4166`:

```
if (consensusParams.NetworkUpgradeActive(nHeight, Consensus::UPGRADE_CANOPY)) {
    // Funding streams are checked inside ContextualCheckTransaction.
} else if ...
```

This reads as if something was missing, and as such it could use a better explanation regarding why no operation is performed. Alternatively, consider specifying `return` instead of doing nothing.

- The Funding Stream validation implemented performs less error checking than that of the example implementation.

While it is understood that the example implementation is not exactly a reference, parameter validation is less stringent in the current implementation. More specifically, the code does not seem to perform the following checks regarding the values of the numerator and denominator at any point.

```
assert(valueNumerator < valueDenominator);
assert(valueNumerator < INT64_MAX / MAX_MONEY);
```

ZIP 251

ZIP 251 defines the deployment of the Network Upgrade 4 (NU4), also known as the Canopy network upgrade.

- The Canopy Testnet activation height in the protocol specification on page 54 is still defined as a **TODO**:

$$\text{CanopyActivationHeight} : \mathbb{N} := \begin{cases} 1046400, & \text{for } \textit{Mainnet} \\ \text{TODO :}, & \text{for } \textit{Testnet} \end{cases}$$

However, this height has been defined, as can be seen in ZIP 251:

```
ACTIVATION_HEIGHT (Canopy)
    Testnet: 1028500

    Mainnet: 1046400
```

- Since the Canopy activation height has been decided, it is unclear why it is not set in the implementation. For example, in `src/chainparams.cpp`, there are several instances of the following piece of code:

```
consensus.vUpgrades[Consensus::UPGRADE_CANOPY].nActivationHeight =
            Consensus::NetworkUpgrade::NO_ACTIVATION_HEIGHT;
```

Additionally, there exists a `TODO` related to the Canopy activation height in `src/chainparams.cpp` on line 153, where a comment states:

```
// TODO: This `if` can be removed once canopy activation height is set.
```

- The link on line 45 of `src/consensus/upgrades.cpp` does not currently point to any valid webpage.

```
/*.strInfo =*/ "See https://z.cash/upgrade/canopy/ for details.",
```

ZIP 211

ZIP 211 aims to deprecate Sprout shielded transactions by removing the ability to add new value to the Sprout value pool balance.

- In order to prevent performing unauthorized transactions, the function `z_mergetoaddress()` in `src/wallet/rpcwallet.cpp` introduced the variable `isFromNonSprout`. Initially set to `false`, this variable is set to `true` if the sending address is not a Sprout address, through a series of conditional statements. At the end of the corresponding function, this variable is checked and if set to `true` when the destination address is a Sprout address, an exception will be raised. An example of this behavior is excerpted in the code below.

```
bool isFromNonSprout = false;
...
if (address == "ANY_TADDR") {
    useAnyUTXO = true;
    isFromNonSprout = true;
} else if (address == "ANY_SPROUT") {
    useAnySprout = true;
} else if (address == "ANY_SAPLING") {
    useAnySapling = true;
    isFromNonSprout = true;
...
if (canopyActive && isFromNonSprout && isToSproutZaddr) {
    // Value can be moved  within Sprout, but not into Sprout.
    throw JSONRPCError(RPC_VERIFY_REJECTED, "Sprout shielding is not supported after Canopy");
}
```

While this code currently performs as expected, in the event that some code is added later and the variable `isFromNonSprout` was not set to `true`, transactions *into* Sprout would be allowed.

In this case, consider changing the logic such that the default value raises the exception if it goes through.

ZIP 212

ZIP 212 introduces changes to the *Note* format in order to avoid having to rely on the soundness of the underlying zk-SNARK to ensure the privacy properties of Sapling shielded transactions. Practically, this change replaces the `rcm` field of *note plaintext* (as defined in section 5.5 Encodings of Note Plaintexts and Memo Fields in the protocol specification) with a field `rseed`, from which field elements are derived with a Pseudorandom Function (PRF). This new note format also updates the `leadbyte` from `0x01` to `0x02`, which will take effect after the introduction of the Canopy upgrade. ZIP 212 also introduces a *grace period* in order to facilitate the transition to the new note format.

- In practice, the introduced changes require some modifications of the decryption logic, based on the `leadbyte` and to accommodate the grace period. More specifically, before the introduction of the Canopy upgrade, all Sapling notes should have a `leadbyte` equal to `0x01`. Once the upgrade is activated, and before the end of the grace

period, the `leadbyte` values `0x01` and `0x02` are both accepted, but after the end of the grace period, only `0x02` will be accepted.

Most of the relevant logic is performed in the `plaintext_version_is_valid()` in `src/zcash/Note.hpp` and by the function of the same name in `zcash_primitives/src/note_encryption.rs` for `librustzcash`.

The correct computation of this value was a specific focus area and the table below was a helpful visual representation during our investigation. The computation performed in the `plaintext_version_is_valid()` function matches the requirements of the ZIP and of the protocol specification.

leadbyte	$<$ Canopy	[Canopy, GracePeriod)	$\geq$ GracePeriod
0x01	Valid	Valid	Invalid
0x02	Invalid	Valid	Valid
other	Invalid	Invalid	Invalid

As discussed above, the introduction of ZIP 212 adds complexity and a number of edge cases that now have to be specifically handled. Throughout the code base, comments have been added to call out specific areas that have been modified with the implementation of ZIP 212.

Although some effort was put into describing these recent changes, consider providing more explicit comments and pointers to the ZIP or to the protocol specification. We provide a few examples below.

A first example concerns the following statement taken from ZIP 212:

> After the activation of this ZIP, any Sapling output of a coinbase transaction that is decrypted to a note plaintext as specified in (10), MUST have note plaintext lead byte equal to `0x02`.
>
> This applies even during the "grace period", and also applies to funding stream outputs (9) sent to shielded payment addresses, if there are any.

These two sentences actually describe a number of different cases. In the implementation, this check is performed in `src/main.cpp`, which seems to cover all cases encompassed by the two requirements. The relevant code block is excerpted below.

```
// ZIP 212: Check that the note plaintexts use the v2 note plaintext
// version.
// This check compels miners to switch to the new plaintext version
// and overrides the grace period in plaintext_version_is_valid()
if (canopyActive != (encPlaintext->get_leadbyte() == 0x02)) {
    return state.DoS(DOS_LEVEL_BLOCK,
        error("CheckTransaction(): coinbase output description has invalid note plaintext version"),
        REJECT_INVALID, "bad-cb-output-desc-invalid-note-plaintext-version");
}
```

In that case, consider explicitly calling out the requirements in the ZIP, since they seem more complex than what the implementation makes them appear.

Another example can be found in `src/zcash/Note.cpp`, where the consistency of the ephemeral key pair is verified. The following check is performed in the function `plaintext_checks_without_height()`, which is called when decrypting a note:

```
// Check that epk is consistent with esk
...
if (expected_epk != epk) {
    return boost::none;
```

```
}
```

It may be valuable to add that this check is a requirement in the ZIP, namely:

> the recipient MUST compute esk as $\mathsf{ToScalar}(\mathsf{PRF}^{\mathsf{expand}}_{\mathsf{rseed}}([5]))$ and check that $\mathsf{epk} = [\mathsf{esk}]\mathsf{g}_\mathsf{d}$ and fail decryption if this check is not satisfied.

This also applies to `librustzcash`, where for example, a similar consistency check of the ephemeral public key is performed in `zcash_primitives/src/note_encryption.rs`.

```
if let Some(derived_esk) = note.derive_esk() {
    if note.g_d.mul(derived_esk, &JUBJUB) != *epk {
        return None;
    }
}
```

A final example is the following comment in `rpcwallet.cpp`, in the function `z_viewtransaction()`:

```
// We don't need to check the leadbyte here: if wtx exists in
// the wallet, it must have already passed the leadbyte check
```

It appears that the following requirement in ZIP 212 refers to this particular case:

> If the plaintext lead byte is not accepted, then the note MUST be discarded. However, if an implementation decrypted the note from a mempool transaction and it was accepted at that time, but it is later mined in a block after the end of the grace period, then it MAY be retained.

Again, consider being slightly more explicit.

- In function `plaintext_checks_without_height()` in `src/zcash/Note.cpp`, the following check is performed at the very end of the function. It may be advisable to move it to the beginning of the function, closer to the other consistency checks, and thus to fail prior to computing the commitment.

```
if (plaintext.get_leadbyte() != 0x01) {
    // ZIP 212: Additionally check that the esk provided to this function
    // is consistent with the esk we can derive
    if (esk != plaintext.generate_or_derive_esk()) {
        return boost::none;
    }
}
```

Update: this item was discussed with the Zcash team. The position of this check follows the protocol specification and is thus correct as is.

- To generate the ephemeral secret key `esk` and the `rcm` element, the value `rseed` is used as input to a PRF. Good cryptographic practices are followed, since distinct diversifiers (namely `4` and `5`) are used for the two values, as can be seen in the code excerpted below:

```
uint256 PRF_rcm(const uint256& rseed)
{
    uint256 rcm;
    auto tmp = PRF_expand(rseed, 4);
    librustzcash_to_scalar(tmp.data(), rcm.begin());
    return rcm;
}
```

```
uint256 PRF_esk(const uint256& rseed)
{
    uint256 esk;
    auto tmp = PRF_expand(rseed, 5);
    librustzcash_to_scalar(tmp.data(), esk.begin());
    return esk;
}
```

However, it would be good practice to specify these diversifiers in the `src/zcash/prf.h`, thereby also reducing code duplication.

- With the introduction of ZIP 212, a few instances of the following pattern can be observed throughout the code base:

```
if (leadbyte != 0x01)
    //  post ZIP 212 operation
else
    // pre ZIP 212 operation
```

This specific pattern might be dangerous when introducing new upgrades conflicting with operations introduced with ZIP 212.

This has also been noted by developers. For example, several comments asking about the safety of this practice were raised in the corresponding Pull Request, for instance in `src/zcash/Note.cpp`:

```
uint256 SaplingNotePlaintext::generate_or_derive_esk() const {
    if (leadbyte != 0x01) {
        return PRF_esk(rseed);
    } else {
        uint256 esk;
        // Pick random esk
        librustzcash_sapling_generate_r(esk.begin());
        return esk;
    }
}
```

Consider following a more future-proof approach, by following the example of the `SerializationOp` function in `src/zcash/Note.hpp`:

```
if (leadbyte != 0x01 && leadbyte != 0x02) {
    throw std::ios_base::failure("lead byte of SaplingNotePlaintext is not recognized");
}
```

- In `librustzcash`, there seems to be mixed type definitions of the block height, sometimes as a signed or unsigned integer.

For example, in the function `scan_block` in `zcash_client_backend/src/welding_rig.rs`, the `scan_output` function is called by casting the height to an unsigned value.

```
if let Some(output) = scan_output::<P>(
    block.height as u32,
    to_scan,
```

In comparison, a few other places define heights as signed integers, for example in `zcash_client_sqlite/src/lib.rs`:

```
#[cfg(feature = "mainnet")]
const SAPLING_ACTIVATION_HEIGHT: i32 = 419_200;
```

or in the function `validate_combined_chain()` in `zcash_client_sqlite/src/chain.rs`, which returns:

```
(block.height as i32, block.prev_hash())
```

Note that in the `zcash` project, the height seems to be more consistently defined as a signed integer. For example, in `zcash/src/chain.h`, the height is defined as follows:

```
//! height of the entry in the chain. The genesis block has height 0
int nHeight;
```

However, in `src/wallet/rpcwallet.cpp`, a transaction's expiry height is cast to an `int64_t`, even thought the quantity `nExpiryHeight` is defined as a `const uint32_t nExpiryHeight` in `src/primitives/transaction.h`.

```
entry.pushKV("expiryheight", (int64_t)wtx.nExpiryHeight);
```

Given the reasonable values the `height` variable can take, it does not seem that this could result in an issue.

ZIP 215

ZIP 215 introduces clear and consistent validity criteria for the decoding of Ed25519 curve points. It also updates the verification equation for JoinSplitSig, and relaxes some constraints about the canonical encoding of points. This change affects the `zcash` project and the underlying `ed25519-zebra` library, which is used post Canopy to perform relevant cryptographic operations.

- In order to ensure the correctness of the underlying verification function, a comprehensive set of test vectors (168 public key/signature pairs in total) has been defined in `src/gtest/test_consensus.cpp`. These test vectors contain encoded curve points that either trigger an error when decoded with libsodium and succeed with the `ed25519-zebra` library, or that pass both decoding functions.

  Interestingly, one of the invalid points identified in finding NCC-ZCHX006-001 on page 23, namely the point with a y coordinate equal to 1 and $x = 0$ with a non-zero expected least significant bit, is not contained within the set of test vectors. For reference, this point is encoded as follows:

  ```
  0100000000000000000000000000000000000000000000000000000000000080
  ```

  Though missing, this point is correctly accepted by `ed25519-zebra`.

  NCC Group also noticed that `ed25519-zebra` defines a test (`conformance` in `tests/small_order.rs`) which seems to be used to generate and display all the test vectors used in `test_consensus.cpp`. This test generates 196 different test vectors, but `test_consensus.cpp` only has 168 of these test cases, which means that 28 are missing. It would be recommended to address this discrepancy, or at least to add test vectors containing the above invalid encoded point.

- In `src/crypto/common.h`, the function `init_and_check_sodium()` initializes and checks that the version of libsodium is the correct one.

  ```
  // What follows is a runtime test that ensures the version of libsodium
  // we're linked against checks that signatures are canonical (s < L).
  ```

  This might have to be revisited with the introduction of ZIP 215 and the change to the `ed25519-zebra` library.

Additionally, there are still a few instances in the code base where direct calls to the libsodium verification function `crypto_sign_verify_detached` are performed, for example in `src/transaction_builder.cpp`:

```
// Sanity check Sprout joinSplitSig
if (crypto_sign_verify_detached(
    mtx.joinSplitSig.data(),
    dataToBeSigned.begin(), 32,
    mtx.joinSplitPubKey.begin()) != 0)
{
    return TransactionBuilderResult("Sprout joinSplitSig sanity check failed");
}
```

Since the function `librustzcash_zebra_crypto_sign_verify_detached` is more permissive than the libsodium function, it is unclear why these function calls have not been replaced by calls to the `ed25519-zebra` library, provided that Canopy is active.

Finally, the comment on line `1049` of `src/main.cpp` will be outdated with the introduction of the Canopy upgrade:

```
// We rely on libsodium to check that the signature is canonical.
// https://github.com/jedisct1/libsodium/commit/62911edb7ff2275cccd74bf1c8aefcc4d76924e0
if (ed25519_verifier(&tx.joinSplitSig[0],
                        dataToBeSigned.begin(), 32,
                        tx.joinSplitPubKey.begin()
                        ) != 0) {
```

General Observations

In this section we present general observations related to the code base and programming practices.

- In `miner.cpp`, the function `SetFoundersRewardAndGetMinerValue()` computes the reward for the miner. It is unclear whether it is intentional, but `miner_reward + nFees` will be returned even if `height <= 0`:

```
if (nHeight > 0) {
    if (chainparams.GetConsensus().NetworkUpgradeActive(nHeight, Consensus::UPGRADE_CANOPY)) {
        ...
    } else if (nHeight <= chainparams.GetConsensus().GetLastFoundersRewardBlockHeight(nHeight)) {
        ...
    } else {
        // Founders reward ends without replacement if Canopy is not activated by the
        // last Founders' Reward block height + 1.
    }
}

return miner_reward + nFees;
```

- In the protocol specification, under the *References* section, Henry de Valence's name is spelled Henry de Valance in reference [ZIP-215].
- There are a couple of useless semicolons, more frequently after function definitions, for example in `src/consensus/funding.cpp` and a few unused functions, such as `RecoverSaplingNote()` in `src/wallet/wallet.cpp`. Both of these have been spotted by the developers.
- There are a few instances of obscure variable naming, for example in `src/main.cpp`, on line `1052`, where the message to be verified is called `dataToBeSigned`:

```
if (ed25519_verifier(&tx.joinSplitSig[0],
                        dataToBeSigned.begin(), 32,
                        tx.joinSplitPubKey.begin()
                        ) != 0) {
```

Another example can be found in `src/wallet/wallet.cpp`, in the `DecryptSaplingNote()` function where the variables could use some more descriptive names.

```
auto output = this->vShieldedOutput[op.n];
auto nd = this->mapSaplingNoteData.at(op);
...
assert(maybe_pt != boost::none);
auto notePt = maybe_pt.get();

auto maybe_pa = nd.ivk.address(notePt.d);
assert(maybe_pa != boost::none);
auto pa = maybe_pa.get();

return std::make_pair(notePt, pa);
```

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 B on page 31.

Title	Status	ID	Risk
Implementation of Ed25519 Signature Verification Does Not Match Protocol	Fixed	001	Low
Missing Braces after `if` Statement	Fixed	002	Low

Finding	**Implementation of Ed25519 Signature Verification Does Not Match Protocol**
Risk	**Low** Impact: None, Exploitability: Low
Identifier	NCC-ZCHX006-001
Status	Fixed
Category	Cryptography
Component	Ed25519
Location	Protocol section 5.4.5, and `zcash/depends/patches/libsodium/`
Impact	The mismatch between implementation and protocol can be used to break consensus between different implementations.
Description	The JoinSplitSig signature scheme is a derivative of Ed25519; it is used in Zcash to sign transactions that contain at least one JoinSplit description. For Ed25519 signature verification, the main implementation[11] uses libsodium[12] in its version 1.0.18, with two extra patches[13] that more closely emulate the behavior of version 1.0.15 of libsodium, which the Zcash implementation initially used. The Zcash protocol[14] (section 5.4.5) includes provisions that describe the exact behavior that is expected from implementations; all implementations should accept or reject given signatures and public keys identically, since this is part of the consensus rules. Any public key or signature treated differently by distinct implementations may lead to a breach of consensus and an unwanted fork.

Notations: below is a list of the notations used in the description that follows:

- Ed25519 uses curve Edwards25519, where points have coordinates (x, y), with x and y being elements of the field $\mathbb{Z}_p$ of integers modulo the prime $p = 2^{255} - 19$.
- The curve has order 8ℓ, with ℓ being a known prime slightly greater than 2^{252}. That curve contains a subgroup of order ℓ; one element of that subgroup, called B, is a fixed conventional generator for that subgroup.
- The curve also contains a cyclic subgroup of order 8. If we conventionally name T one generator of that group, then the contents of the subgroup are exactly:
  - $0T = (0, 1)$, of order 1 (this is the neutral element of the addition law on curve points);
  - $4T = (0, p - 1)$, of order 2;
  - $2T$ and $6T$, of order 4;
  - T, $3T$, $5T$ and $7T$, of order 8.
- Any point P on the curve can be uniquely decomposed into the sum of a point in the subgroup of order ℓ, and a point in the subgroup of order 8. The latter is called the *low-order component* of the point P. Points in the subgroup generated by B are exactly the points whose low-order component is the neutral element $(0, 1)$.
- A *secret key* is a non-zero integer a modulo ℓ (i.e. $0 < a < \ell$). The corresponding public key

[11] https://github.com/zcash/zcash
[12] https://github.com/jedisct1/libsodium
[13] https://github.com/zcash/zcash/tree/552482a404c1de2912db4273898ce2c2d8990ad7/depends/patches/libsodium
[14] https://github.com/zcash/zips/blob/564d7f630ec156847aa6ee08e8e630f98fde5e8f/protocol/protocol.pdf

is point $A = aB$.

- A *signature* on a message m is a pair (R, s) where R is a curve point, and s an integer modulo ℓ, that fulfills the verification equation $sB = R + kA$, where k is the SHA-512 hash (reduced modulo ℓ) of a string that includes R, A, and the signed message m.

Encodings: curve points are encoded over strings of exactly 32 bytes, such that:

- The first 255 bits are the little-endian encoding of the y coordinate of the point (nominally in the $0...p-1$ range).
- The 256-th bit (which is the most significant bit of the last byte of the string) is a copy of the least significant bit of the x coordinate.

This is known as a *compressed* format, since it does not include the full encodings of both coordinates x and y. Using the curve equation, the square x^2 of the x coordinate can be recomputed from the y coordinate, and the provided least significant bit of x can be used to determine which of the two square roots of x^2 is the actual x coordinate. This format is used both for public keys (A) and for the first half of each signature (R).

Zcash protocol rules: section 5.4.5 of the Zcash protocol states that, for purposes of decoding points (A and R) the specifications in the original Ed25519 article[15] are followed, with two extra rules:

- Strings that are part of a specific list of eleven excluded 32-byte strings (called "ExcludedPointEncodings" in the protocol) are rejected.
- The y coordinate, nominally in the $0...p-1$ range, is allowed to be greater, and will be reduced modulo p automatically (since y is encoded over 255 bits, and $p = 2^{255} - 19$, this means that the nineteen values from $2^{255} - 19$ to $2^{255} - 1$ are acceptable). Encodings of y with integers equal to or greater than p are said to be *non-canonical*.

The list of excluded strings is *meant* to be the list of low-order points; it has more than eight entries because of possible non-canonical encodings. However, this is not exactly the case: the list contains some points that are *not* of low order, and conversely does not contain some encodings that lead to points of low order. There appears to be a historical confusion, inherited from earlier versions of libsodium, between 256-bit encodings of points, and 256-bit encodings of integers. Indeed, the list in the Zcash protocol seems to consider the whole sequence of 256 bits to be an integer in the $0..2^{256} - 1$ range, thus potentially up to $2p + 37$. However, the actual encoding uses only 255 bits for y, not 256. The consequence is that the list contains four sequences that are not, in fact, encodings of low-order points:

```
13e8958fc2b227b045c3f489f2ef98f0d5dfac05d3c63339b13802886d53fc85
daffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
b4176a703d4dd84fba3c0b760d10670f2a2053fa2c39ccc64ec7fd7792ac03fa
d9ffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

The first two of these strings are valid encodings of points which happen to be of order 8ℓ (thus, these points cannot appear as part of non-maliciously generated public keys and signatures, and it is not "wrong" to reject them). The other two are not decodable as points, and would have been rejected anyway by libsodium.

Conversely, the following low-order points are *not* rejected by the implementation:

[15] https://ed25519.cr.yp.to/ed25519-20110926.pdf

```
0000000000000000000000000000000000000000000000000000000000000080
26e8958fc2b227b045c3f489f2ef98f0d5dfac05d3c63339b13802886d53fc85
c7176a703d4dd84fba3c0b760d10670f2a2053fa2c39ccc64ec7fd7792ac03fa
edffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
0100000000000000000000000000000000000000000000000000000000000080
ecffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
eeffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffff
```

In practice, accepting or rejecting low-order points does not induce vulnerabilities with regard to the signature functionality: the signature incarnates consent of the signer, a non-malicious signer will not generate a low-order public key, and use of low-order R points does not give any advantage for signature forgeries. The existence of low-order points not covered by the exclusion list is alluded to in the non-normative note at the end of section 5.4.5 of the protocol; however, there is an additional issue here. Consider the last three of the points above: they encode a y coordinate equal to 1 or $p-1$ modulo p (the last point encodes $p+1$, which is a non-canonical encoding of 1), leading to $x^2 = 0$, and thus $x = 0$ (unique possible square root of zero); but these encodings also specify that the least significant bit of x should be 1. There is, nominally, no possible solution. These points are still accepted by the Zcash implementation; indeed, in the final steps of point decompression, the value of x is adjusted as follows (in function `ge25519_frombytes_negate_vartime()`):

```
    if (fe25519_isnegative(h->X) == (s[31] >> 7)) {
        fe25519_neg(h->X, h->X);
    }
```

This means that the recomputed x is negated if its least significant bit does not match the expected value, but nothing checks that the least significant bit of $-x$ is correct. When $x = 0$, the least significant bit of x is 0, and so is the least significant bit of $-x = 0$.

The Zcash protocol relies on the original Ed25519 article, but that article does not, actually, specify exact encoding rules. In particular, it says nothing about what to do when the recomputed x^2 is equal to 0, but the expected least significant bit of x is 1. The original "ref10" implementation from the article authors accepts the flawed encoding; this behavior was inherited by libsodium and thus by the Zcash implementation. Another, more precise source for Ed25519 is RFC 8032,[16] and it explicitly rejects these encodings (in section 5.1.3, clause 4).

The lack of specification of how that specific situation ($x = 0$ but lsb of x is 1) should be handled can arguably be called a defect in the protocol document. In particular, the mathematically correct behavior (rejecting the point, since there is no solution matching the received encoding) is *not* the one followed by the implementation, but is mandated by RFC 8032.

The Zcash implementation deviates from the above description in two situations:

- The list of excluded points in the Zcash implementation has *twelve* entries, not eleven. The twelfth entry (point `dbffff...ff`) is rejected by the implementation, but is not in the list specified in the protocol. In practice, this string is the canonical encoding of a valid point U of order 4ℓ. Since that point has order greater than ℓ, it is not in the subgroup generated by B, and it cannot appear in a non-maliciously generated public key or signature.

  Moreover, the discrete logarithm in base B of the component of order ℓ of U is not known and should be computationally infeasible to recover; without that discrete logarithm, it

[16] https://tools.ietf.org/html/rfc8032

does not seem to be possible to craft a public key and a signature that would match the verification equation $sB = R+kA$ with point $R = U$. Therefore, while there mathematically exist values A, s and m such that (U, s) would embody the discrepancy between protocol rules and implementation ((U, s) is an acceptable signature on m as per the protocol rules, but the implementation would reject it), there is no known way to practically compute such values A, s and m.

- While the 12-entry list of excluded strings is applied by the Zcash implementation for decoding points R in signatures, it is *not* applied for public keys. Instead, a simple test is performed to reject the all-zeros sequence:

```
unsigned char d = 0;
for (int i = 0; i < 32; ++i) {
    d |= pk[i];
}
if (d == 0) {
    return -1;
}
```

This means that any public key whose encoding is part of the 11-entry excluded point list in the Zcash protocol section 5.4.5, but is not the all-zeros string, will be deemed acceptable by the Zcash implementation. In particular, the encoding of the curve neutral element (`0100..00`) allows making signatures (R, s) where $R = sB$ that will be accepted as valid for *any* message m.

Summary: three specific flaws have been described here:

- The protocol specification does not cover the case of $x = 0$ with a non-zero expected least significant bit. The current Zcash implementation accepts this case, but it is explicitly rejected by RFC 8032.
- The Zcash implementation rejects a specific point R in signatures, that should be deemed acceptable as per the protocol (such signatures mathematically exist, but won't be generated by non-malicious actors, and seem infeasible to generate in practice).
- The Zcash implementation accepts signatures and public keys that the protocol rejects as invalid.

The third flaw is most likely to lead to consensus breaches, since it can be easily exercised. However, it should be fixed by the application of ZIP 215, which will make all points, including low-order points, acceptable as public keys or signatures. On the other hand, the first flaw (lack of specification for the invalid $x = 0, \text{lsb}(x) = 1$ situation) will not be fixed by ZIP 215 as currently worded.

Recommendation

Since the protocol is supposed to match the implementation, and the implementation cannot be retroactively fixed, the protocol will have to be amended in order to match the reality of the situation:

- All sequences of bits that can be decoded as points, except the all-zeros strings, are acceptable as public keys.
- The missing twelfth string should be added to the ExcludedPointEncodings list.
- If, during point decompression, the recomputed x coordinate is zero, then the specified least significant bit value is ignored.

When NU4 is activated, as part of the new rules described by ZIP 215, most of these issues disappear, since all low-order points become acceptable, both as public keys and in signatures. The situation about $x = 0$ must still be documented, though, since that specific point decoding issue remains relevant post-NU4.

Finding **Missing Braces after `if` Statement**

Risk **Low** Impact: Low, Exploitability: Low

Identifier NCC-ZCHX006-002

Status Fixed

Category Data Validation

Component Wallet

Location `src/wallet/rpcwallet.cpp`

Impact A statement expected to be reached under specific conditions is now unconditionally performed, resulting in potentially unexpected behavior.

Description The upcoming deployment of the "Canopy" upgrade to the Zcash network resulted in the introduction of a number of changes to the implementation of the Zcash protocol. Since the project is being actively used while the update has not been deployed yet, the code frequently follows the following pattern in order to seamlessly transition to the updated specifications once "Canopy" is in effect.

```
if (canopyActive) {
    // perform canopy-specific action
}
```

There is an instance in the function `zc_raw_joinsplit()` in `src/wallet/rpcwallet.cpp` where this `canopyActive` check was added after a *single statement `if` block*, without subsequently surrounding it by curly braces, as can be seen in the following code excerpt.

```
3044 if (params[3].get_real() != 0.0)
3045     if (canopyActive) {
3046         throw JSONRPCError(RPC_VERIFY_REJECTED,
         ➔  "Sprout shielding is not supported after Canopy");
3047     }
3048     vpub_old = AmountFromValue(params[3]);
```

Thus, the statement `vpub_old = AmountFromValue(params[3])` was effectively *kicked out* of the conditional test and is now reached unconditionally.

The function `zc_raw_joinsplit()` is invoked through the Zcash RPC client `zcash-cli`, via the `zcrawjoinsplit` option. It is used to create new raw JoinSplit transactions and is now deprecated in favor of `z_sendmany`. The `vpub_old` variable is populated by one of the command-line arguments and represents an amount to be moved into the confidential value store.

In practice, this oversight does not seem to have any consequence. Prior to this modification, the variable was only set when the command-line argument was non-zero. The change in the logic now sets `vpub_old` regardless. However, since the variable `vpub_old` is initialized to `0` earlier in the function, the ramifications seem non-existent (at least as long as the `get_real` and `AmountFromValue` functions perform correctly).

Recommendation Add curly braces after the first `if` statement, as in the example below.

```
if (params[3].get_real() != 0.0) {
    if (canopyActive) {
```

```
        throw JSONRPCError(RPC_VERIFY_REJECTED,
        ➜  "Sprout shielding is not supported after Canopy");
    }
    vpub_old = AmountFromValue(params[3]);
}
```

Appendix A: ZIP Reference Details

This audit covered the specification (Phase 1) and implementation (Phase 2) of the following six ZIPs:

Title	Specification	Implementation
ZIP 214 – Consensus rules for a Zcash Development Fund	https://zips.z.cash/zip-0214	https://github.com/zcash/zcash/pull/4560
ZIP 207 – Funding Streams	https://zips.z.cash/zip-0207	https://github.com/zcash/zcash/pull/4560
ZIP 251 – Deployment of the Canopy Network Upgrade	https://zips.z.cash/zip-0251	https://github.com/zcash/zcash/pull/4487
ZIP 211 – Disabling Addition of New Value to the Sprout Value Pool	https://zips.z.cash/zip-0211	https://github.com/zcash/zcash/pull/4489
ZIP 212 – Allow Recipient to Derive Sapling Ephemeral Secret from Note Plaintext	https://zips.z.cash/zip-0212	https://github.com/zcash/zcash/pull/4578, https://github.com/zcash/librustzcash/pull/258
ZIP 215 – Explicitly Defining and Modifying Ed25519 Validation Rules	https://zips.z.cash/zip-0215	https://github.com/zcash/zcash/pull/4581, https://github.com/ZcashFoundation/ed25519-zebra/pull/24

The pull requests identified in the Implementation column above are part of the v3.1.0 release of Zcash (https://github.com/zcash/zcash/tree/v3.1.0). The implementation review also included changes to the `ed25519-zebra` library to implement the validation rules of ZIP 215.

Two versions of the Zcash Protocol Specification, dating from August 3rd, 2020 (version 2020.1.12, https://github.com/zcash/zips/blob/fb64b2e4303b332ef8960fc6bbf34b0598596c5d/protocol/protocol.pdf) and August 11th, 2020 (version 2020.1.13, https://github.com/zcash/zips/blob/6e5278ed95e334dc861838da26857d1c0dcf638f/protocol/protocol.pdf), which included modifications introduced by the above ZIPs, were also used as a references for phase 2.

Appendix B: 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.